UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

PROGRAMACIÓN CON INDETERMINISMO: UN ENFOQUE BASADO EN REESCRITURA.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Juan Rodríguez Hortalá

Bajo la dirección de los doctores

Francisco J. López Fraguas
Jaime Sánchez Hernández

Madrid, 2010

ISBN: 978-84-693-8793-1

Programación con indeterminismo: un enfoque basado en reescritura

Juan Rodríguez Hortalá

Tesis doctoral

Directores: Francisco J. López Fraguas
Jaime Sánchez Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Abril de 2010

Programming with Non-Determinism: a Rewriting Based Approach

Juan Rodríguez Hortalá

PhD Thesis

Advisors: Francisco J. López Fraguas
Jaime Sánchez Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

April 2010

"¿Y la ciencia? La ciencia es, ante todo, resignación. Normalmente sólo se resaltan sus logros, pero éstos llegan con lentitud y no igualan nunca la enormidad de sus fracasos. La ciencia es la aceptación de la mortalidad y la arbitrariedad del individuo, que surge del equilibrado juego de los espermatozoides en su lucha por lograr la primacía en la fecundación. Es el reconocimiento del transcurrir, de la irreversibilidad, de la ausencia de recompensa, de una justicia superior, del conocimiento absoluto, de la comprensión global; sería incluso heroica si sus creadores no fueran tan a menudo ignorantes de lo que están haciendo en realidad."

Stanisław Lem — Más relatos del piloto Pirx (Opowieści o pilocie Pirxie) - 1968

"And science? Science is, above all, resignation. Usually, only its achievements are highlighted, but these come slowly and cannot ever match the enormity of its failures. Science is the acceptance of mortality and the arbitrariness of the individual, emerging from the balanced game of the spermatozoids in their struggle for primacy in fertilization. It is the recognition of the passage of time, of irreversibility, of the lack of rewards, higher justice, absolute knowledge and global understanding ; it could even be heroic if its creators were not so often ignorant of what they are actually doing."

Stanisław Lem — More Tales of Pirx the Pilot (Opowieści o pilocie Pirxie) - 1968

Agradecimientos

Esta tesis no hubiera sido posible sin la ayuda y el apoyo de muchas personas. Primero quisiera dar las gracias a mis directores por la confianza depositada y el esfuerzo dedicado en sacar adelante el trabajo que constituye esta tesis. Gracias también a Michael Hanus y a los miembros del Institut für Informatik de la Christian-Albrechts-Universität zu Kiel—especialmente a Sebastian Fischer, Bernd Braßel y Jan Christiansen—, por su hospitalidad durante mi estancia en Kiel, así como por las interesantes discusiones allí mantenidas. Asimismo, gracias a Stephan Merz y a los miembros del Loria por el amable trato dispensado durante mi estancia en Nancy, y por ayudarme a empezar con el sistema Isabelle.

Gracias a los revisores anónimos por ayudarnos a depurar nuestros artículos, darnos a conocer referencias enriquecedoras, y proponernos nuevas líneas de investigación o aplicaciones diferentes de nuestro trabajo. También debo agradecer el apoyo económico en forma de becas, contratos, y de los proyectos de investigación en los que he participado, recibido por parte de varias instituciones: el Ministerio de Educación y Ciencia de España, la Consejería de Educación de la Comunidad de Madrid, y la Universidad Complutense de Madrid.

Gracias a mis compañeros de la Complutense por los consejos y la experiencia que me han transmitido durante estos años. Gracias a los miembros del Grupo de Programación Declarativa y a toda la gente del 220—entre los que incluyo al Dr. Álvez—, por hacer muchos más llevaderos los días malos en el trabajo, y ayudarme a no desfallecer. Gracias a mi familia por haberme llevado hasta aquí. Gracias a mis amigos por hacer que tenga vida y proyectos fuera del trabajo, especialmente a David por poder contar siempre con él. Y finalmente gracias también a María por todas esas cosas y todas las demás.

Acknowledgements

This thesis would not have been possible without the help and support of many people. First I would like to thank my advisors for their trust and the effort put in the development of the work that constitutes this thesis. Thanks also to Michael Hanus and the members of the Institut für Informatik of the Christian-Albrechts-Universität zu Kiel—especially Sebastian Fischer, Bernd Brassel and Jan Christiansen—, for their hospitality during my stay in Kiel, and for the interesting discussions. Likewise thanks to Stephan Merz and the members of Loria for the kind treatment during my stay in Nancy, and for helping me to start with the Isabelle system.

Thanks to the anonymous referees for helping us to refine our papers, for giving us to know a lot of enriching references, and for proposing new research lines or different applications of our work. Also I acknowledge the financial support in the form of grants, contracts, and research projects I have participated, received from various institutions: the Ministry of Education and Science of Spain / Ministerio de Educación y Ciencia de España, the Ministry of Education of the Regional of Madrid / Consejería de Educación de la Comunidad de Madrid and the Complutense University of Madrid / Universidad Complutense de Madrid.

Thanks to my colleagues at the Complutense for the advice and experience passed on me during these years. Thanks to the members of the Declarative Programming Group and all the people from office 220—among which I include the Dr. Álvez—, for making the bad days at work much easier to bear, and for helping me not to faint. Thanks to my family for bringing me here. Thanks to my friends for giving me a life outside the office, specially to David because I can always count on him. And finally thanks to María for all those things and all the others.

Foreword

This PhD thesis has been developed under the "publications format" recently adopted for the presentation of PhD thesis in Spanish universities, which determines its structure: according to this format a PhD can consist of a collection of papers previously published in relevant venues, together with an extended summary of the research performed. In Part I we give the summary itself. Part II contains the list of publications that constitute this thesis, accompanied by the either the full text for each publication as they were originally published, or a link to its electronic edition—for copyrighted publications. Finally, in Part III we include the extended versions of some of the most important publications, where the interested reader can find the complete proofs of our results.

In particular, Part I begins in Chapter 1 with a presentation of the subject of the thesis, the use of non-determinism in programming. After that introduction to the context of our work we are ready to state the goals of this thesis, and its concrete structure. Then follows Chapter 2, which is just a translation to Spanish of Chapter 1.

Chapter 3 is the longest one, and consists of an overview of the work developed in the papers that constitute the thesis, trying to organize and integrate them to form a coherent piece. After a small review of the state of the art in the field non-deterministic programming—in the sense exposed in Sect. 1.1—we may find three sections, each corresponding to one of the three fundamental semantic alternatives for non-determinism that have been considered in this work. Each subsection deals with a particular subject, that may have been treated in one or more of our papers. We have chosen to present results from different papers in the same subsection instead of inspecting the publications one by one, because our aim in this chapter is to get a standalone overview of our work whose reading could be useful by itself, with no need to resort to the papers. This way a reader interested in the details about a particular subject could then go to the referred papers for further information. Hence, for each subject we proceed by motivating it, then we give the minimal technical notions needed, and present the main technical results and maybe some particularly interesting intermediate results; finally some conclusions or related and future work are discussed. Formal proofs have been avoided along this entire Part, and can be found in the corresponding publications or extended versions.

When writing this document we faced the problem that the publications of Part II—that constitute the core of the thesis but have been written along several years—do not always follow the same notational conventions, or differ in the formulation of some notions and results. This has compelled us to establish a compromise between readibility of the summary in Part I and readability of the publications themshelves. In some cases, we have decided to stick to the publication conventions, and therefore the reader may find some minor changes in the notation conventions between some chapters of the summary. In other cases, we have decided to slightly change the presentation of the material with respect to the original works. We hope our decisions have helped to improve the overall legibility of this thesis.

To conclude, in Chapter 4 we summarize the general conclusions achieved in this thesis, and propose some possible lines of future work. The corresponding translation to Spanish can be found in Chapter 5.

Prólogo

Esta tesis doctoral ha sido desarrollada bajo el "formato de publicaciones" recientemente adoptado para la presentación de tesis doctorales en las universidades españolas, lo que ha determinado su estructura: de acuerdo a este formato una tesis doctoral puede consistir en una colección de artículos previamente publicados en congresos o revistas relevantes, junto con un extenso resumen de la investigación realizada. En la parte I podemos encontrar el resumen propiamente dicho. La parte II contiene la lista de publicaciones que constituyen la tesis, acompañando a cada publicación, bien el texto completo correspondiente tal y como fué publicada originalmente, o bien un enlace a su edición electrónica en el caso de publicaciones cuyos derechos de reproducción han sido cedidos a alguna editorial. Finalmente, en la parte III incluimos las versiones extendidas de algunas de las publicaciones más importantes, donde el lector interesado podrá encontrar las demostraciones completas de cada uno de nuestros resultados.

En particular la parte I comienza en el capítulo 1 con una presentación del tema de la tesis: el uso del indeterminismo en programación. Después de esta introducción al contexto de nuestro trabajo ya estamos preparados para establecer los objetivos de la tesis, así como su estructura concreta. Tras esto encontramos el capítulo 2, que no es más que una traducción al castellano del capítulo 1.

El capítulo 3 es el más largo de todos, y consiste en una panorámica del trabajo desarrollado en los artículos que conforman la tesis, tratando de organizarlos e integrarlos para que formen una pieza coherente. Después de una pequeña revisión del estado del arte en el campo de la programación indeterminista—en el sentido expuesto en la sección 1.1—podemos encontrar tres secciones, una por cada una de las tres alternativas semánticas fundamentales para el indeterminismo que hemos considerado en este trabajo. A su vez cada subsección trata un tema particular, que puede haber sido desarrollado en uno o más de nuestros artículos. Hemos decidido permitirnos presentar resultados de trabajos diferentes en una misma subsección en lugar de examinar las publicaciones una a una, porque nuestra intención en este capítulo es obtener una panorámica autocontenida de nuestro trabajo cuya lectura pudiera ser útil por sí misma, sin necesidad de recurrir a los artículos. De esta manera un lector interesado en los detalles acerca de una cuestión en particular podría entonces dirigirse a los artículos referidos para más información. Por tanto para cada materia procedemos empezando por motivarla, después damos la mínimas nociones técnicas necesarias y seguidamente presentamos los principales resultados técnicos obtenidos, a veces acompañados por algunos resultados intermedios interesantes; finalmente terminamos con unas breves conclusiones o una pequeña discusión sobre trabajo relacionado o futuro. Las demostraciones formales han sido evitadas a lo largo de esta parte de la tesis, y pueden ser encontradas en sus publicaciones correspondientes, o en las versiones extendidas de estas.

Durante la redacción de este documento nos hemos encontrado con el problema de que las publicaciones de la parte II—que aunque constituyen el núcleo de la tesis, han sido desarrollados a lo largo de varios años—no siempre siguen las mismas convenciones de notación, o bien difieren en la formulación de algunas nociones y resultados. Esto nos ha obligado a establecer un compromiso entre la legibilidad del resumen de la parte I y la legibilidad de las publicaciones en sí mismas. En algunos casos hemos preferido ceñirnos a las convenciones establecidas en las publicaciones, por lo que el lector podrá encontrar algunos

cambios menores en las convenciones de notación utilizadas en los diferentes capítulos del resumen. En otros hemos decidido modificar ligeramente la presentación del material respecto a los trabajos originales. Esperamos que estas decisiones hayan ayudado a mejorar la legibilidad conjunta de la tesis.

Para terminar, en el capítulo 4 resumimos las conclusiones generales obtenidas a lo largo de esta tesis, y proponemos algunas posibles líneas de trabajo futuro. La correspondiente traducción al castellano puede encontrarse en el capítulo 5.

Contents

List of Figures

Part I

Summary of the Research

Chapter 1

Introduction

"Causal determinism is, roughly speaking, the idea that every event is necessitated by antecedent events and conditions together with the laws of nature. The idea is ancient, but first became subject to clarification and mathematical analysis in the eighteenth century. Determinism is deeply connected with our understanding of the physical sciences and their explanatory ambitions, on the one hand, and with our views about human free action on the other. In both of these general areas there is no agreement over whether determinism is true (or even whether it can be known true or false), and what the import for human agency would be in either case."

Stanford Encyclopedia of Philosophy — Causal Determinism - 2008

1.1 Non-determinism and Functional Logic Programming

In the field of programming languages it is usual to consider that a *language* is *deterministic* when the evaluation of a given expression always computes the same values for the same input data. Hence most of imperative languages (Pascal, C/C++, Java, C#, ...) are considered deterministic. Nevertheless it is very easy to build a program in any of these languages that does not fulfil this condition. On one hand we can generate a random number or just check the system time and present it as the output of a program without arguments, thus getting different results for different runs of the program. Thus determinism is lost in the interaction with the operative system, or the outer world, which is not explicitly modeled in the programs. On the other hand we may build a concurrent program, where the order in which each forked process ends its corresponding task cannot be predicted, and then generate an output depending on the order of finalization of processes.

But the kind of non-determinism that we study here is not that which arises from the interaction with the outside, nor due to concurrent processes, nor even because of the physical limitations of hardware. This work is about **using non-determinism as a language expressive feature**. In non-deterministic languages some primitives or other resources are provided to express computations whose final result is not totally determined by the input data. In these languages, in which concurrency does not need to be present, non-determinism is part of the computation model.

Different variants of non-determinism have been used since a long time in system specification (e.g., Turing Machines or non-deterministic automata) or for programming (the constructions of McCarthy [McC63] or Dijkstra [Dij97] are classical examples). Non-determinism is especially useful for searching problems, because it allows to express them

in a natural and direct way: each alternative path is expressed in the language through a non-deterministic branching, and the system handles by itself the exploration of the different possibilities. Nevertheless in deterministic languages it is the programmer who has to manually explicitly encode in her algorithm the exploration of the different alternatives, usually a non trivial and error prone task.

For its great expressive power, the use of non-determinism has aroused special interest in the field of declarative programming. In particular it is at the basis of the Prolog language [SS86], maybe the most famous representative of non-deterministic languages. Functional languages, also in the declarative programming field, usually do not consider non-determinism as a language feature, but it is not stranger to their interest [McC63, SS92, HM95, KSS98, LM99, SSH00], and several works have been developed to simulate it by different constructions [Wad85, Hin00, KSFS05, FBK05, NAR07, FKS09].

Functional-logic programming (**FLP**) [Rod01, Han07, Han94], o more generally, multiparadigm declarative programming, constitutes an important research field, that tries to integrate in the same language the main virtues of several independent paradigms: logic programming, lazy functional programming, and even constraint programming. Two modern representatives of this line are the languages *Toy* [LS99, CSe06] and *Curry* [Han06], which share their main features. In these languages, non-confluent term rewriting systems (**TRS's**) [BN98, TeR03] are used as programs to support non-strict non-deterministic functions, which are one of the most distinctive features of the paradigm [GHLR99, AH02]. Those TRS's follow the constructor discipline also, corresponding to a value-based semantic view, in which the purpose of computations is to produce values made of constructors, that we call constructor terms or simply **c-terms**. The operational mechanism of FLP languages is based on narrowing [Han07], an extension of term rewriting that replaces matching by unification [BN98].

TRS's already had a long tradition as a suitable basic formalism to address a wide range of tasks in computer science, in particular, several specification languages [CDE+07, FN97, vdBMR02], theorem provers [WP06, CPR06], transformation tools [BKVV08], and other programming languages besides FLP languages [Pla95, PJ87] are based on TRS's. In some of these applications, that elegant way of expressing non-determinism through the use of a non-confluent TRS is exploited again to obtain a clean and high level representation of complex systems.

Therefore non-confluent constructor-based TRS's, also called constructor systems (**CS's**), can be used as a common syntactic framework for FLP and rewriting. The set of rewrite rules constitutes a program and so we also call them *program rules*. Nevertheless the behaviour of current implementations of FLP and rewriting differ substantially, because the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions, that were explored in [SS92]. In that work the different language variants that result after adding non-determinism to a basic functional language are expounded, structuring the comparison as a choice among different options over several dimensions: strict/non-strict functions, angelic/demonic/erratic non-deterministic choices and *singular/plural semantics* for parameter passing. As usual in the mainstream of FLP, through this work we will assume non-strict angelic non-determinism, so we are concerned about the last dimension only. The key difference is that under a singular semantics, in the substitutions used to instantiate the program rules for function application, the variables of the program rules should range over single objects of the set of values considered; in a plu-

ral semantics those range over sets of objects. This has been traditionally identified with the distinction between *call-time choice* and *run-time choice* [Hus93] parameter passing mechanisms. Under call-time choice a value for each argument is computed before performing parameter passing, this corresponds to call-by-value [Plo75] in a strict setting and to call-by-need [AFM+95, AF97, MOW98] in a non-strict setting , in which a partial value instead of a total value is computed. On the other hand, run-time-choice corresponds to call-by-name [Plo75], each argument is copied without any evaluation and so the different copies of any argument may evolve in different ways afterwards. Thus, traditionally it has been considered that call-time choice parameter passing induces a singular semantics while run-time choice induces a plural semantics.

Example 1.1.1. Consider the program $\mathcal{P} = \{f(c(X)) \to d(X,X), X\ ?\ Y \to X, X\ ?\ Y \to Y\}$. With call-time choice/singular semantics to compute a value for the term $f(c(0?1))$ we must first compute a (partial) value for $c(0?1)$, and then we may continue the computation with $f(c(0))$ or $f(c(1))$ which yield $d(0,0)$ or $d(1,1)$. Note that $d(0,1)$ and $d(1,0)$ are not correct values for $f(c(0?1))$ in that setting.
On the other hand with run-time choice/plural semantics to evaluate the term $f(c(0?1))$:

- Under the run-time choice point of view, the step $f(c(0?1)) \to d(0?1, 0?1)$ is sound, hence not only $d(0,0)$ and $d(1,1)$ but also $d(0,1)$ and $d(1,0)$ are valid values for $f(c(0?1))$.
- Under the plural semantics point of view, we consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0?1)$ in which every element matches the argument pattern $c(X)$. Therefore the set $\{0,1\}$ can be used for parameter passing obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$, which evaluation yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$ and $d(1,0)$.

In general, call-time choice/singular semantics produces less results than run-time choice/ plural semantics.

A standard formulation for call-time choice[1] in FLP is the *CRWL*[2] logic [GHLR96, GHLR99], which is implemented by current FLP languages like *Toy* [LS99, CSe06] or *Curry* [Han06]; on the other hand term rewriting is considered the standard semantics for run-time choice[3], and is the basis for the semantics of non-deterministic specification languages like Maude [CDE+07], CafeOBJ [FN97] or Elan [vdBMR02], but has been rarely [Ant97] thought as a valuable global alternative to call-time choice for the value-based view of FLP. However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal. Two different approaches for the combination of call-time choice and run-time choices were proposed by us in [LRS09a, LRS09c]. The advantage of those approaches is that both call-time choice and run-time choice have clean and high level formulations—the *CRWL* logic and term rewriting, respectively—which are also consolidated frameworks, thus making programs written using that semantics combination easy to understand, at least for a reader used to declarative programming or to formal methods in general. This is also stressed by the fact that the proposals of [LRS09a, LRS09c] are conservative extensions of either pure run-time choice or pure call-time choice.

[1] In fact angelic non-strict call-time choice.
[2] **C**onstructor-based **R**e**W**riting **L**ogic.
[3] In fact angelic non-strict run-time choice.

Nevertheless the use of an operational notion like term rewriting as the semantic basis of a FLP language can lead us to confusing situations, not very compatible with the value-based semantic view that we wanted for the constructor-based TRS's used in FLP.

Example 1.1.2. Starting with the program of Example 1.1.1 we want to evaluate the expression $f(c(0)\ ?\ c(1))$ with run-time choice/plural semantics:

- Under the run-time choice point of view, that is, using term rewriting, the evaluation of the subexpression $c(0)?c(1)$ is needed in order to get an expression that matches the left hand side $f(c(X))$. Hence the derivations $f(c(0)?c(1)) \to f(c(0)) \to d(0,0)$ and $f(c(0)?c(1)) \to f(c(1)) \to d(1,1)$ are sound and compute the values $d(0,0)$ and $d(1,1)$, but neither $d(0,1)$ nor $d(1,0)$ are correct values for $f(c(0)?c(1))$.

- Under the plural semantics point of view, we consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0)?c(1)$ in which every element matches the argument pattern $c(X)$. Therefore the set $\{0,1\}$ can be used for parameter passing obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$ that yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$ and $d(1,0)$.

Which of these is the more suitable perspective for FLP?

This problem did not appear in [SS92] because no pattern matching was present, nor in [Hus93] because only call-time choice was adopted (and this conflict does not appear between the call-time choice and the singular semantics views). Choosing the run-time choice perspective of term rewriting has some unpleasant consequences. First of all the expression $f(c(0?1))$ has more values than the expression $f(c(0)?c(1))$, even when the only difference between them is the subexpressions $c(0?1)$ and $c(0)?c(1)$, which have the same values both in call-time choice, run-time choice and plural semantics. This is pretty incompatible with the value-based semantic view we are looking for in FLP. And this has to do with the loss of some desirable properties, present in *CRWL*, when switching to run-time choice. In [Rod08] the authors introduced π*CRWL*, a variation of *CRWL* to express plural semantics, and showed how plural semantics recovers those properties, which are very useful for reasoning about computations. In a later work [RR09b] the implementation of this semantics in the Maude system was presented. In both works, we have tried to show how π*CRWL* allows natural encodings of some programs that need to do some collecting work, while returning to the value-based view of programmig that was lost in the run-time choice alternative.

1.2 Goals and Structure of the Work

1.2.1 Goals

In this work we have tried to make some contributions to the field of non-deterministic functional-logic programming. Our objectives are diverse, often at the level of semantic descriptions, where we are looking to provide constructions and results that hopefully could be of use to deepen in the understanding of the meaning of programs, or for program manipulation, analysis and transformation. We are also concerned with more practical aspects, and some prototypes have been developed as a consequence. Sometimes we are placed in a consolidated framework—namely call-time choice or run-time choice—while at

other times we decided to explore the expressive possibilities of non-deterministic functions by proposing new semantic frameworks, some of them coming from the combination of several semantics, others presenting more novel semantics proposals.

We have considered **two general goals**.

1. **Providing new descriptions of existing semantics for non-determinism in TRS's**. Having different equivalent semantic descriptions for the same formalism, at different abstraction levels, can be useful because it allows us to turn to the more suitable point of view when reasoning about a particular aspect of that formalism. Classical examples are the full abstraction of [Plo77] or the semantic triad denotational-operational-verification calculus of [Str06]. This motivates our interest in providing new equivalent semantic formulations for the classical semantic proposals for TRS's, that is, call-time choice and run-time choice.

   a) *For call-time choice*: emphasis on *rewriting-like operational models*, as *CRWL* already provides a declarative semantics for call-time choice.
   As we saw before, term rewriting is not a suitable framework for describing the semantics of modern FLP because it induces a run-time choice semantics instead of the intended call-time choice semantics. Nevertheless term rewriting is a very good formalism for describing computations, as it provides a precise, simple and high abstraction level notion of step in the process of reducing an expression to its associated value. Therefore we will try to devise a modification of term rewriting that would be sound for call-time choice while retaining the simplicity and elegance of term rewriting.

   b) *For run-time choice*: emphasis in *declarative semantics*, as term rewriting already provides a simple notion of reduction step for run-time choice.
   Hence we want to define a rewriting logic that could play the same role for term rewriting that *CRWL* would do for the novel rewriting notion of item *a)*. Although Meseguer's rewriting logic [MM02] was already conceived a long time ago and has been used in many works to reason about term rewriting computations, we want our new logic to focus on constructor-based TRS's only, so this more specialized tool could be more suitable for certain kind of reasoning about these CS's, which are particularly interesting for its use in the description of programming languages. In short, what we want is to characterize the set of c-terms reachable by term rewriting from a given expression, and to do it in a compositional way wrt. some sensible notion of semantic value.

   The interest in the following subjects also arises naturally after considering the previous goals.

   - *Connecting several semantic descriptions of modern functional-logic programming*: Besides *CRWL*, there are some other families of semantic descriptions for the call-time choice semantics implemented in modern FLP systems. All those formalisms are expected to describe the same intended semantics, as it is tacitly accepted in the FLP community. Nevertheless it would be nice to have precise technical results about their equivalence at our disposal, because these could be used to share techniques and transfer results between those frameworks. This

way, as each formalism establishes its own model of FLP from a different perspective and abstraction level, we could address each future problem regarding analysis or transformation of FLP programs from the most suitable perspective.

That is the main motivation for our first general goal, hence we find interesting trying to contribute to develop a unified outlook of the semantic descriptions for modern FLP.

- *Studying the full abstraction problem for non-deterministic rewriting-based languages*: Given a semantics and a notion of operational observation, that semantics is fully abstract wrt. that observable whenever two expressions have the same semantics if and only if they are observationally indistinguishable. Henceforth full abstraction indicates a perfect correspondence between the semantics and the actual behavior of program pieces.

  In items *a)* and *b)* we have already seen some frameworks in which we may find a logical semantics accompanied by an adequate operational counterpart, thereby that operational notion is an obvious candidate to be used to define our observations. Hence it is natural to study the full abstraction problem for that frameworks, and so it is one of the goals of our work.

- *Experimenting with the use of automatic theorem provers or proof assistants for reasoning about the semantics of functional-logic programming*: In the last years there is a growing trend [ABF+05] stating that the combination of formal semantics and mechanized theorem proving will be ubiquitous or at least very important in the future programming languages technology. This combination could be fruitful in at least two different ways. First of all, it could help to debug the formalization of the semantics and theory of programming languages, and to provide new insights or clarify overlooked aspects. Besides, the formalization in these mechanized provers could be used to implement program analysis or transformations, whose soundness would be certified by the mechanized prover itself, thus leading to the development of certified program manipulation tools.

  We also think that this line would be coherent with our first general goal, as it would provide new instruments for reasoning about programs. For all these reasons we find interesting trying to contribute to this line, to be more precise, to the mechanized formalization of the semantics of FLP.

2. **Investigating new semantics alternatives**.

   a) *Semantic combinations*: We have already seen in Sect. 1.1 that the combination call-time choice and run-time choice semantics in the same language can be interesting to implement certain programming patterns, so we will try to propose some alternatives for this combination. For each of them we would like to precisely formalize its intended semantics, find some representative examples of the possibilities of the new language, and if possible, provide a prototype for experimenting with it.

   b) *A plural semantics with pattern matching*: Although this cannot be strictly considered some a priori goal of the research that constitutes this thesis, the plural semantics formalized through the $\pi CRWL$ proof calculus, which was informally introduced at the end of Sect. 1.1, came up naturally through our searching for a new rewriting logic for term rewriting. After this discovery, our duties to

this semantics became clear easily: studying its properties, relating it with the previous semantics of call-time choice and run-time choice, and finding some clues about its possible implementation.

1.2.2 Structure of the Work

This PhD thesis consists of several parts. Part I is a summary of the research performed during the development of this thesis, so it starts with and introduction to the subject of the thesis, and a presentation of the goals of this work, as we have seen. Then follows Chapter 2, which is just a translation to Spanish of Chapter 1. After that in Chapter 3 we first make a small review of the state of the art in the field (Sect. 3.1), followed by three other sections, each of them presenting the results obtained for one of the three semantic alternatives informally presented in Sect. 1.1.

Henceforth in Sect. 3.2 we present the advances regarding *call-time choice semantics*: the equivalence between *CRWL* and the operational semantics of [AHH+05] (Sect. 3.2.1, [LRS07a]); a novel operational notion called *let*-rewriting, in which the notions of subexpressions sharing and call-time choice are added to the framework of term rewriting (Sect. 3.2.2, [LRS07b]); its associated *let*-narrowing relation (Sect. 3.2.3, [LRS09d]) and the extensions of both notions to higher order (Sect. 3.2.4, [LRS08a]); the problem of full abstraction of *CRWL* wrt. *let*-rewriting in their higher order versions (Sect. 3.2.5, [LR10]); and finally our first approach to the formalization of *CRWL* in the Isabelle theorem prover (Sect. 3.2.6, [LMR09a]).

On the other hand, in Sect. 3.3 we can find our results concerning *run-time choice semantics*: the proposal of a new rewriting logic for classical term rewriting that defines a semantics that is fully abstract wrt. natural observational notions using term rewriting as their operational procedure (Sect. 3.3.1, [LRS09b]); a comparison between call-time choice and run-time choice regarding the set of computed c-terms, showing that call-time choice computes strictly less values in general and exactly the same values for deterministic programs, and its extension to their narrowing versions (Sect. 3.3.2, [LRS07b, LRS09d]); and two different proposals for the combination of call-time choice and run-time choice in the same language, either in a run-time choice environment or in a call-time choice environment (Sect. 3.3.3, [LRS09a, LRS09c]).

Finally, in Sect. 3.4 we show the first results about the novel *plural semantics* outlined above: its formalization through the proof calculus $\pi CRWL$, some of its basic properties and the extension of the comparison of Sect. 3.3.2 now taking into account this new plural semantics (Sect. 3.4.1, [Rod08]); a program transformation to implement $\pi CRWL$ in run-time choice, and the use of this transformation to develop a prototype for $\pi CRWL$ in the Maude system (Sect. 3.4.2, [Rod08, RR09b]).

The last chapter of this first part is Chapter 4, in which we present our conclusions, summarizing the contributions of our work and evaluating our achievements wrt. the starting goals. This chapter—and so this part—concludes with some considerations about the research lines opened for a possible future development. The corresponding translation to Spanish can be found in Chapter 5.

Part II begins with Chapter 6, in which we have listed the publications that constitute this thesis. Then follows Chapter 7, that contains for each publication either the full text as they were originally published or a link to its electronic edition—for copyrighted publications—, and which concludes this part. The last Part III consists of a single Chapter

8, in which we may find the extended versions of some of the most important publications of the thesis.

Chapter 2

Introducción

"El determinismo causal es, a grandes rasgos, la idea de que cada evento se produce de forma necesaria debido a los eventos y condiciones que lo preceden, junto con las leyes de la naturaleza. La idea es antigua, pero no fue hasta el siglo XVIII que ésta se convirtió en objeto de aclaración y análisis matemático. El determinismo está profundamente relacionado con nuestra comprensión de las ciencias físicas y con sus ambiciones explicativas, por un lado, y con nuestro punto de vista acerca de la libertad de acción humana, por el otro. En ninguna de estas dos áreas generales existe un acuerdo sobre si dicho determinismo acontece en realidad (ni siquiera acerca de si realmente es posible establecer la certeza o falsedad de dicha afirmación), y qué impacto en la acción humana tendría en cualquier caso."

Stanford Encyclopedia of Philosophy — Causal Determinism - 2008

2.1 El indeterminismo y la programación lógico-funcional

En el ámbito de los lenguajes de programación es habitual considerar que un *lenguaje* es *determinista* cuando la evaluación de una expresión dada siempre calcula los mismos valores para la misma configuración de sus valores de entrada. Así la mayoría de los lenguajes imperativos (Pascal, C/C++, Java, C#, ...) son considerados deterministas. Sin embargo es muy sencillo diseñar un programa en cualquiera de aquellos lenguajes para el que ya no se cumpla dicha condición. Por un lado podemos generar números aleatorios o simplemente obtener la hora actual del reloj del sistema y presentar cualquiera de ellos como la salida de un programa sin argumentos de entrada, obteniendo por tanto resultados diferentes para distintas ejecuciones del programa. De esta forma el determinismo se pierde durante la interacción con el sistema operativo, o "el mundo exterior", que no es modelado explícitamente en los programas. Por otra parte también podemos escribir un programa concurrente para el que el orden en el que cada proceso concurrente termina su tarea correspondiente no pueda ser predicho, y además definir la salida del programa de forma que esta sea dependiente del orden de finalización de dichos procesos.

Sin embargo el tipo de indeterminismo que estudiamos en este trabajo no es aquel que surge de la interacción con el exterior, ni el debido a la ejecución de procesos concurrentes, ni siquiera el causado por las limitaciones físicas del *hardware*. Este trabajo trata acerca del **uso del indeterminismo como un recurso expresivo del lenguaje**. En los lenguajes

indeterministas se ofrecen algunas primitivas u otros recursos que pueden ser utilizados para expresar cómputos cuyo resultado final no está totalmente determinado por los datos de entrada. En estos lenguajes, en los que la concurrencia no tiene porqué estar presente, el indeterminismo es parte del modelo de cómputo.
Distintas variantes del indeterminismo han sido empleadas desde hace mucho tiempo en la especificación de sistemas (por ejemplo las máquinas de Turing o los autómatas indeterministas) o en programación (las construcciones de McCarthy [McC63] o Dijkstra [Dij97] son ejemplos clásicos). El indeterminismo es especialmente útil para la resolución de problemas de búsqueda, debido a que permite expresar este tipo de problemas de una forma natural y directa: cada camino alternativo es expresado en el lenguaje mediante una alternativa indeterminista, y es el sistema el que se encarga de manejar por sí mismo el proceso de exploración de las diferentes posibilidades. En un lenguaje determinista, en cambio, es el programador el que esta obligado a codificar explícitamente en su algoritmo dicha exploración de las distintas alternativas, una tarea habitualmente no trivial y proclive a errores.

Por sus grandes capacidades expresivas, el uso del indeterminismo ha despertado un especial interés en el campo de la programación declarativa. En particular este está en la base del lenguaje Prolog [SS86], quizás el más famoso representante de los lenguajes indeterministas. Los lenguajes funcionales, también el campo de la programación declarativa, no suelen considerar el indeterminismo como una característica del lenguaje, aunque este no es ajeno a su interés [McC63, SS92, HM95, KSS98, LM99, SSH00], y en varios trabajos se ha tratado su simulación mediante diferentes construcciones [Wad85, Hin00, KSFS05, FBK05, NAR07, FKS09].

La programación lógico-funcional (**FLP**) [Rod01, Han07, Han94], o en general, la programación declarativa multi-paradigma, constituye un importante campo de investigación que intenta integrar en el mismo lenguaje las principales virtudes de varios paradigmas independientes: programación lógica, programación funcional perezosa e incluso programación con restricciones. Dos representantes modernos de esta linea son los lenguajes *Toy* [LS99, CSe06] y *Curry* [Han06], que comparten sus características principales. En estos lenguajes se emplean sistemas de reescritura de términos (**TRS's**) [BN98, TeR03] no confluentes como programas, de esta manera soportando funciones no estrictas e indeterministas, que son una de las características distintivas del paradigma [GHLR99, AH02]. Dichos TRS's además siguen la disciplina de constructoras, correspondiendo a una visión semántica basada en valores en la que el propósito de los cómputos es producir valores compuestos de constructoras, que llamaremos términos construidos o simplemente **c-términos**. El mecanismo operacional de los lenguajes lógico funcionales está basado en el estrechamiento [Han07], una extensión de la reescritura de términos que reemplaza el ajuste de patrones por la unificación [BN98].

Los sistemas de reescritura tienen ya una larga tradición como un formalismo básico apropiado para abordar un amplio rango de tareas en las ciencias de la computación. En particular varios lenguajes de especificación [CDE+07, FN97, vdBMR02], demostradores de teoremas [WP06, CPR06], herramientas de transformación de programas [BKVV08] y otros lenguajes de programación además de los lógico-funcionales [Pla95, PJ87] están basados en TRS's. En algunas de estas aplicaciones se explota una vez más esa forma elegante de expresar el indeterminismo mediante el uso de un TRS no confluente, para obtener una representación limpia y de un alto nivel de abstracción de sistemas complejos.

Por tanto los sistemas de reescritura basados en constructoras posiblemente no confluentes, también llamados sistemas de constructoras (**CS's**), pueden usarse como un marco sintáctico común de la FLP y la reescritura de términos. El conjunto de reglas de reescritura constituye el programa, por tanto las llamamos *reglas de programa*. Sin embargo el comportamiento de las implementaciones actuales de FLP y de la reescritura difieren sustancialmente, porque la introducción del indeterminismo en un lenguaje funcional da lugar a una variedad de decisiones semánticas, que ya fueron exploradas en [SS92]. En dicho trabajo se exponen las diferentes variantes de lenguaje que resultan de la adición de una primitiva para el indeterminismo a un lenguaje funcional básico, estructurando la comparación como una elección entre diferentes opciones sobre varias dimensiones: funciones estrictas o no estrictas, elecciones indeterministas angélicas, demoníacas o erráticas, y una *semántica singular* o *plural* para el paso de parámetros. Como es habitual en la corriente principal de la programación lógico-funcional, a lo largo de este trabajo asumiremos indeterminismo no estricto y angélico, por lo que sólo nos preocuparemos por la última dimensión. La diferencia clave es que bajo una semántica singular, en las sustituciones empleadas para instanciar las reglas de programa para la aplicación de una función, las variables de las reglas de programa deben ser instanciadas con objetos individuales del conjunto de valores considerado; en una semántica plural estas deben ser instanciadas con conjuntos de objetos. Tradicionalmente esta dicotomía ha sido identificada con la distinción entre los mecanismos de paso de parámetro con *call-time choice* o con *run-time choice* [Hus93]. Bajo call-time choice un valor para cada argumento es calculado antes de realizar el paso de parámetros, lo que corresponde con call-by-value [Plo75] en un entorno de funciones estrictas, y con call-by-need [AFM+95, AF97, MOW98] en un entorno no estricto, en el que un valor parcial en vez de total es calculado. Por otra parte run-time choice corresponde a call-by-name [Plo75], en el que cada argumento se copia sin ninguna evaluación por lo que diferentes copias de cualquier argumento pueden evolucionar de forma independiente más adelante. De esta forma, tradicionalmente se ha considerado que el paso de parámetros con call-time choice induce una semántica singular, mientras que el paso de parámetros con run-time choice induce una semántica plural.

Example 2.1.1. Consideremos el programa $\mathcal{P} = \{f(c(X)) \to d(X, X), X \,?\, Y \to X, X \,?\, Y \to Y\}$. Bajo call-time choice/semántica singular para calcular un valor para el término $f(c(0?1))$ primero debemos calcular un valor (parcial) para $c(0?1)$, y entonces ya podemos continuar el cómputo con $f(c(0))$ o con $f(c(1))$, que resultan en $d(0, 0)$ o $d(1, 1)$. Nótese que de esta manera $d(0, 1)$ y $d(1, 0)$ no son valores correctos para $f(c(0?1))$ en dicho entorno. Por otra parte bajo run-time choice/semántica plural para evaluar el término $f(c(0?1))$:

- Bajo el punto de vista de run-time choice, el paso $f(c(0?1)) \to d(0?1, 0?1)$ es correcto, por lo que no sólo $d(0, 0)$ y $d(1, 1)$ sino también tanto $d(0, 1)$ como $d(1, 0)$ son valores válidos para $f(c(0?1))$.

- Bajo el punto de vista de una semántica plural podemos considerar el conjunto $\{c(0), c(1)\}$, que es un subconjunto del conjunto de valores para $c(0?1)$ en el que cada elemento encaja con el patrón $c(X)$ del argumento de f. Así el conjunto $\{0, 1\}$ puede usarse para el paso de parámetros obteniendo una especie de "expresión conjuntista" $d(\{0, 1\}, \{0, 1\})$, con lo que la evaluación resulta en los valores $d(0, 0)$, $d(1, 1)$, $d(0, 1)$ y $d(1, 0)$.

En general el call-time choice/semántica singular produce menos resultados que el run-time choice/semántica plural.

Una formulación estándar del call-time choice[1] en FLP es la lógica *CRWL*[2] [GHLR96, GHLR99], que se implementa en los lenguajes FLP actuales como *Toy* [LS99, CSe06] o *Curry* [Han06]; por otra parte la reescritura de términos está considerada como la semántica estándar para run-time choice[3], y es la base de la semántica de algunos lenguajes indeterministas de especificación como Maude [CDE+07], CafeOBJ [FN97] o Elan [vdBMR02], pero rara vez [Ant97] ha sido considerada una alternativa global válida al call-time choice para la visión basada en valores de la FLP. Sin embargo, puede haber partes de un programa o funciones individuales para las que el run-time choice pudiera ser una mejor opción, y por tanto sería conveniente dejar ambas opciones (run-time/call-time) disponibles para el programador. Dos enfoques diferentes para la combinación de call-time choice y run-time choice fueron propuestas por nosotros en [LRS09a, LRS09c]. La ventaja de dichos enfoques es que tanto call-time choice como run-time choice disponen de formulaciones limpias y de alto nivel de abstracción—la lógica *CRWL* y la reescritura de términos, respectivamente—que además son marcos consolidados, consiguiendo por tanto que los programas escritos en esa combinación semántica sean fáciles de entender, al menos para un lector acostumbrado a la programación declarativa o a los métodos formales en general. Esto queda también subrayado por el hecho de que las propuestas de [LRS09a, LRS09c] sean extensiones conservadoras bien del run-time choice puro, o del call-time choice puro.

Sin embargo el uso de una noción operacional como la reescritura de términos como la fundamentación semántica de un lenguaje FLP nos puede conducir a situaciones confusas, no muy compatibles con la visión basada en valores que deseamos para los sistemas de constructoras usados en FLP.

Example 2.1.2. Tomando el programa del ejemplo 2.1.1, queremos evaluar la expresión $f(c(0)\ ?\ c(1))$ bajo run-time choice/semántica plural:

- Bajo el punto de vista de run-time choice, es decir, usando reescritura de términos, la evaluación de la subexpresión $c(0)?c(1)$ es necesaria para conseguir una expresión que encaje con el lado izquierdo de $f(c(X))$. Por tanto las derivaciones $f(c(0)?c(1)) \to f(c(0)) \to d(0,0)$ y $f(c(0)?c(1)) \to f(c(1)) \to d(1,1)$ son correctas y calculan los valores $d(0,0)$ y $d(1,1)$, pero ni $d(0,1)$ ni $d(1,0)$ son valores correctos para $f(c(0)?c(1))$.

- Bajo el punto de vista de una semántica plural podemos considerar el conjunto $\{c(0), c(1)\}$, que es un subconjunto del conjunto de valores para $c(0?1)$ en el que cada elemento encaja con el patrón $c(X)$ del argumento de f. Así el conjunto $\{0,1\}$ puede usarse para el paso de parámetros obteniendo una especie de "expresión conjuntista" $d(\{0,1\},\{0,1\})$, con lo que la evaluación resulta en los valores $d(0,0)$, $d(1,1)$, $d(0,1)$ y $d(1,0)$.

¿Cual de estas dos es la perspectiva más adecuada para la FLP?

Este problema no aparecía en [SS92] porque el lenguaje básico considerado no soportaba ajuste de patrones, ni en [Hus93], donde únicamente se adoptaba el call-time choice (y porque este conflicto no aparece entre las visiones del call-time choice y la semántica singular). Elegir la perspectiva de run-time de la reescritura de términos tiene algunas consecuencias indeseables. Para empezar la expresión $f(c(0?1))$ tiene más valores que

[1] En realidad call-time choice angélico y no estricto.
[2] **C**onstructor-based **Re**e**W**riting **L**ogic.
[3] En realidad run-time choice angélico y no estricto.

la expresión $f(c(0)?c(1))$, incluso cuando las únicas diferencias entre ellas son las subexpresiones $c(0?1)$ y $c(0)?c(1)$, que tienen los mismos valores tanto bajo call-time choice como bajo run-time choice o una semántica plural. Esto es bastante incompatible con la visión semántica basada en valores que estábamos buscando para FLP. Y esto tiene que ver con la pérdida de algunas propiedades deseables, presentes en *CRWL*, al pasar a run-time choice. En [Rod08] presentamos $\pi CRWL$, una variación de *CRWL* que expresa una semántica plural, y vimos cómo esta semántica recupera dichas propiedades, que son muy útiles para razonar sobre los cómputos. En un trabajo posterior [RR09b] se muestra la implementación de esta semántica en el sistema Maude. En ambos trabajos, intentamos enseñar como $\pi CRWL$ permite realizar codificaciones naturales de algunos programas que necesitan realizar algún trabajo de recolección, permaneciendo a la vez en la visión de la programación basada en valores que se perdió en la alternativa de run-time choice.

2.2 Objetivos y estructura del trabajo

2.2.1 Objetivos

En este trabajo hemos intentado hacer algunas contribuciones al campo de la programación lógico-funcional indeterminista. Nuestros objetivos son diversos, a menudo al nivel de las descripciones semánticas, donde tratamos de aportar construcciones y resultados que esperamos puedan ser de utilidad para profundizar en la comprensión del significado de los programas, o como herramientas para la manipulación, análisis y transformación de programas. También nos hemos ocupado de aspectos más prácticos, y algunos prototipos han sido desarrollados a consecuencia de ello. Unas veces trabajamos en un marco consolidado—concretamente call-time choice o run-time choice—mientras que otras hemos decidido explorar las capacidades expresivas de las funciones indeterministas proponiendo nuevos marcos semánticos, algunos de ellos surgiendo de la combinación de semánticas ya existentes, otros presentando propuestas semánticas más novedosas.

Hemos considerados **dos objetivos generales**.

1. **Proporcionar nuevas descripciones para semánticas existentes del indeterminismo en TRS's**. Disponer de varias descripciones semánticas del mismo formalismo, trabajando a distintos niveles de abstracción, puede ser útil porque nos permite situarnos en el punto de vista más cómodo en cada razonamiento concreto sobre un aspecto particular del formalismo. Ejemplos clásicos son la full abstraction de [Plo77] o la terna compuesta por una semántica denotacional, una semántica operacional y un cálculo de verificación de [Str06]. Esto motiva nuestro interés en proporcionar nuevas formulaciones semánticas equivalentes para las propuestas semánticas clásicas para TRS's, es decir, call-time choice y run-time choice.

   a) *Para call-time choice*: énfasis en los *modelos operacionales al estilo de la reescritura*, ya que *CRWL* ya proporciona una semántica declarativa para call-time choice.
   Como vimos antes, la reescritura de términos no es un marco apropiado para describir la semántica de la FLP ya que esta induce una semántica de run-time choice en vez de la pretendida semántica de call-time choice. Sin embargo la reescritura es un formalismo muy bueno para describir cómputos, ya que proporciona una noción precisa, simple y de alto nivel de abstracción de lo que

constituye un paso en el proceso de reducción de una expresión a su valor asociado. Por tanto trataremos de diseñar una modificación de la reescritura que sea correcta para el call-time choice y a la vez preserve la simplicidad y elegancia de la reescritura de términos.

b) *Para run-time choice*: énfasis en la *semántica declarativa*, ya que la reescritura ya proporciona una noción simple de paso de reducción para run-time choice. Así lo que pretendemos es definir una lógica de reescritura que pudiera jugar el mismo papel para la reescritura de términos que el que *CRWL* desarrollaría para la nueva noción de reescritura del apartado *a)*. Aunque la lógica de reescritura de Meseguer [MM02] ya fue concebida hace mucho tiempo y ha sido usada en muchos trabajos para razonar acerca de los cómputos de reescritura, queremos que nuestra lógica se centre únicamente en los sistemas de constructoras, de forma que esta herramienta más especializada pudiera ser más apropiada para cierta clase de razonamientos sobre estos CS's, que son particularmente interesantes por su uso en la descripción de los lenguajes de programación. Siendo concisos, lo que queremos es caracterizar el conjunto de c-términos alcanzables mediante reescritura desde una expresión dada, y hacerlo de una forma composicional con respecto a alguna noción de valor semántico con sentido.

El interés en los temas siguientes también ha surgido de forma natural al considerar los objetivos anteriores.

- *Conectar varias descripciones semánticas de la programación lógico-funcional moderna*: Además de *CRWL*, hay otras familias de descripciones para la semántica de call-time choice implementada por los sistemas FLP modernos. Se espera que todos estos formalismos describan la misma semántica pretendida, como se acepta tácitamente en la comunidad FLP. Sin embargo sería interesante disponer de resultados técnicos precisos acerca de su equivalencia, porque estos podrían ser utilizados para compartir técnicas y transferir resultados entre los distintos marcos. De esta forma, como cada formalismo establece su propio modelo de la FLP desde una perspectiva y nivel de abstracción diferente, podríamos enfocar cada problema futuro acerca del análisis o transformación de programas FLP desde la perspectiva más apropiada.

  Como esta es la principal motivación de nuestro primer objetivo general, encontramos interesante intentar contribuir a desarrollar una panorámica unificada de las descripciones semánticas de la FLP moderna.

- *Estudiar el problema de la full abstraction para lenguajes indeterministas basados en reescritura*: Dada una semántica y una noción de observación operacional, dicha semántica es fully abstract con respecto al observable cuando ocurre que dos expresiones tienen la misma semántica si y solo si son observacionalmente indistinguibles. Por tanto la full abstraction indica una correspondencia perfecta entre la semántica y el comportamiento efectivo del programa.

  En los apartados *a)* y *b)* ya hemos visto algunos marcos en los que podemos encontrar una semántica lógica acompañada de una contrapartida operacional adecuada, así que aquellas nociones operacionales son candidatos obvios en los que basar nuestras nociones de observación. De esta manera el estudio del problema de la full abstraction para estos marcos surge de forma natural, y por tanto lo consideramos uno de los objetivos de nuestro trabajo.

- *Experimentar con el uso de demostradores de teoremas o asistentes de demostración para el razonamiento acerca de las semántica de la programación lógico-funcional*: En estos últimos años ha surgido una tendencia creciente [ABF+05] que sostiene que la combinación de las semánticas formales y la demostración de teoremas mecanizada será ubicua o al menos muy importante en la tecnología de lenguajes de programación futura. Esta combinación puede ser provechosa en al menos dos maneras diferentes. Para empezar podría ayudar a depurar la formalización de las semánticas y la teoría de los lenguajes de programación, y a aportar nuevas claves o clarificar aspectos oscuros. Además la formalización de las semánticas realizada en estos demostradores automáticos podría ser empleada para implementar herramientas para el análisis o transformación de programas, cuya corrección estaría certificada por el demostrador automático en sí mismo, dando lugar al desarrollo de herramientas de manipulación de programas certificadas.

  También consideramos que esta linea sería coherente con nuestro primer objetivo general, ya que aportaría nuevos instrumentos para el razonamiento acerca de los programas. Por todas estas razones encontramos interesante tratar de contribuir en esta linea, es decir, siendo más precisos, a la formulación mecanizada de las semántica de la FLP.

2. **Investigar acerca de nuevas alternativas semánticas**.

   a) *Combinación de semánticas*: Ya hemos visto en la sección 2.1 cómo la combinación de call-time choice y run-time choice en el mismo lenguaje puede ser interesante para implementar algunos patrones de programación, así que trataremos de proponer algunas alternativas semánticas que permitan esta combinación. Para cada una de ellas pretendemos formalizar la semántica pretendida de forma precisa, encontrar algunos ejemplos representativos de las posibilidades del nuevo lenguaje resultante, y si es posible, proporcionar un prototipo para experimentar con él.

   b) *Un semántica plural con ajuste de patrones*: Aunque este no puede ser considerado estrictamente un objetivo a priori de la investigación que constituye esta tesis, la semántica plural formalizada a través del cálculo de pruebas $\pi CRWL$, que fue introducida informalmente al final de la sección 2.1, apareció de forma natural durante nuestra búsqueda de una nueva lógica para la reescritura de términos. Después de este descubrimiento, nuestras obligaciones respecto a esta semántica quedaron claras rápidamente: estudiar sus propiedades, relacionarla con las semánticas anteriores de call-time choice y run-time choice, y encontrar algunas pistas acerca de su posible implementación.

2.2.2 Estructura del trabajo

Esta tesis doctoral está compuesta de varias partes. La parte I es un resumen de la investigación realizada durante el desarrollo de esta tesis, por lo que comienza en su capítulo 1 con una introducción al tema de la tesis y una presentación de los objetivos del trabajo. El capítulo siguiente, en el que nos encontramos, no es más que una traducción al castellano del capítulo 1. Más adelante, en capítulo 3 primero hacemos una pequeña revisión del estado del arte en el campo (Sect. 3.1), seguida por otras tres secciones, cada una de

ellas presentando los resultados obtenidos para una de las tres alternativas semánticas presentadas informalmente en la sección 2.1.

Así en la sección 3.2 presentamos los avances conseguidos acerca de la *semántica de call-time choice*: la equivalencia entre *CRWL* y la semántica operacional de [AHH+05] (Sect. 3.2.1, [LRS07a]); una nueva noción operacional llamada *let*-rewriting, en la que las nociones de compartición de subexpresiones y call-time choice son añadidas al marco de la reescritura de términos (Sect. 3.2.2, [LRS07b]); su relación de *let*-estrechamiento asociada (Sect. 3.2.3, [LRS09d]) y la extensión de ambas nociones para tratar el orden superior (Sect. 3.2.4, [LRS08a]); el problema de la full abstraction de *CRWL* con respecto a *let*-rewriting en sus versiones de orden superior (Sect. 3.2.5, [LR10]); y finalmente nuestro primer acercamiento a la formalización de *CRWL* en el demostrador de teoremas Isabelle (Sect. 3.2.6, [LMR09a]).

Por otra parte, en la sección 3.3 podemos encontrar nuestros resultados acerca de la *semántica de run-time choice*: la propuesta de una nueva lógica para reescritura de términos clásica que define una semántica que es fully abstract con respecto a nociones de observación naturales que usan la reescritura de términos como su procedimiento operacional (Sect. 3.3.1, [LRS09b]); una comparación entre las semántica de call-time choice y run-time choice respecto al conjunto de c-términos calculados por ambas, mostrando que call-time choice calcula estrictamente menos valores en general y exactamente los mismos para programas deterministas, y su extensión a sus versiones de estrechamiento (Sect. 3.3.2, [LRS07b, LRS09d]); y dos propuestas diferentes para la combinación de call-time choice y run-time choice en el mismo lenguaje, bien en un entorno de call-time choice o en uno de run-time choice (Sect. 3.3.3, [LRS09a, LRS09c]).

Finalmente, en la sección 3.4 presentamos los primeros resultados acerca de la nueva *semántica plural* esbozada más arriba: su formalización a través del cálculo de pruebas $\pi CRWL$, algunas de sus propiedades básicas y la extensión de la comparación de la sección 3.3.2 ahora tomando en cuenta a esta nueva semántica plural (Sect. 3.4.1, [Rod08]); una transformación de programa para implementar $\pi CRWL$ con run-time choice, y el uso de esta transformación para desarrollar un prototipo para $\pi CRWL$ en el sistema Maude (Sect. 3.4.2, [Rod08, RR09b]).

El último capítulo de esta parte es el capítulo 4, en el que presentamos nuestras conclusiones, resumiendo las contribuciones de nuestro trabajo y evaluando nuestros logros respecto a los objetivos de partida. Esta capítulo—y por tanto esta parte—concluye con algunas consideraciones acerca de las líneas de trabajo abiertas para un posible desarrollo futuro. La correspondiente traducción al castellano puede encontrarse en el capítulo 5.

La parte II comienza con el capítulo 6, en el que hemos listado las publicaciones que constituyen esta tesis. A este le sigue el capítulo 7, que contiene, para cada publicación, bien el texto completo correspondiente tal y como fué publicada originalmente, o bien un enlace a su edición electrónica en el caso de publicaciones cuyos derechos de reproducción han sido cedidos a alguna editorial, y que concluye esta parte. La última parte III consiste en un único capítulo 8, en el que se pueden encontrar las versiones extendidas de algunas de las publicaciones más importantes de la tesis.

Chapter 3

Programming with Non-Determinism: a Rewriting Based Approach

3.1 State of the Art

Now we will make an overview of some of the most important approaches to the description of the different semantic alternatives for rewriting based non-deterministic languages. We give a more detailed presentation for the frameworks that have been approached in this thesis, while for the others we just make a light introduction with several bibliographic pointers for the interested reader.

3.1.1 Term Rewriting Systems

In the present section we will go through the fundamental concepts in the field of term rewriting systems. This is standard material in the literature on the subject, and its presentation here is based on that of [BN98, TeR03].

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions, h for elements of Σ and $x, y, X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$[1]. We also use $h/n \in CS$ (FS resp.) to denote that $h \in CS^n$ (FS^n resp.). The notation $\overline{o}$ stands for tuples of any kind of syntactic objects. Given a set $\mathcal{A}$ we denote by $\mathcal{A}^*$ the set of finite sequences of elements of that set. For any sequence $a_1 \ldots a_n \in \mathcal{A}^*$ and function $f : \mathcal{A} \to \{true, false\}$, by $a_1 \ldots a_n \mid f$ we denote the sequence constructed taking in order every element from $a_1 \ldots a_n$ for which f holds.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*)

[1] During this work we will adopt the conventions of uppercase or lowercase for variable names depending on the section in which they are placed, but always following the same convention originally used in the papers that are treated in the corresponding section.

is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$)[2]. The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing **values**. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. We say that e is syntactically equal to e' and denote it by $e \equiv e'$ whenever e and e' are exactly the same expression (thus $X \equiv X$; $h(e_1, \ldots, e_n) \equiv h(e'_1, \ldots, e'_n)$ iff $\forall i \in \{1, \ldots, n\}.\ e_i \equiv e'_i$). The set of variables occurring in an expression e will be denoted as $var(e)$. We say than an expression e is ground iff $var(e) = \emptyset$. An expression is linear when no variable occurs more than once in it.
We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$; $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

A *position* in an expression is a sequence of natural numbers separated by dots that determines a subexpression of it. By ϵ we denote the empty position, also called the *root position*, which points to the symbol at the top of the expression. Hence $root(X) = X$; $root(h(\overline{e})) = h$. We write $p, q, o, \ldots$ for positions. Then the set $\mathcal{O}(e)$ of positions in $e \in Exp$ is defined as $\epsilon \in \mathcal{O}(e)$; $i.p \in \mathcal{O}(h(e_1, \ldots, e_n))$ if $i \in \{1, \ldots, n\}$ and $p \in \mathcal{O}(e_i)$. Thus $e|_p$ denotes the subexpression of $e \in Exp$ at position $p \in \mathcal{O}(e)$, and is defined as $e|_\epsilon = e$; $h(e_1, \ldots, e_n)|_{i.q} = e_i|_q$. The set $\mathcal{O}(e)$ can be partitioned into the set $\tilde{\mathcal{O}}(e) = \{p \in \mathcal{O}(e) \mid e|_p \notin \mathcal{V}\}$ of non variable positions and the set $\mathcal{O}_\mathcal{V}(e) = \{p \in \mathcal{O}(e) \mid e|_p \in \mathcal{V}\}$ of variable positions.

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ϵ for the identity (or empty) substitution. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some θ_i; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. Given $W \subseteq \mathcal{V}$ we write $\theta|_W$ for the restriction of θ to W, and $\theta|_{\setminus D}$ is a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$ (therefore for all $X \in \mathcal{V}$). We say that e *subsumes* e', and write $e \preceq e'$, if $e\theta \equiv e'$ for some θ. In this case we also say that e' *matches* e with the *matching substitution* θ, and that e' is an *instance* of e. We write $\theta \preceq \theta'$ if $X\theta \preceq X\theta'$ for all variables X and $\theta \preceq \theta'[W]$ if $X\theta \preceq X\theta'$ for all $X \in W$.

A *term rewriting system* $\mathcal{P}$ (TRS) is a set of rewrite rules of the form $l \to r$ where $l, r \in Exp$ and $l \notin \mathcal{V}$. It is usual to impose the additional restriction $var(r) \subseteq var(l)$ over each rewrite rule, as it is done in classical texts like [BN98, TeR03]. Systems not fulfiling this restriction are called *term rewriting system with extra variables*. In particular we say that X is an extra variable in the rule $l \to r$ iff $X \in var(r) \setminus var(l)$, and by $vExtra(R)$ we denote the set of extra variables in a rule R.

Given a TRS $\mathcal{P}$, its associated *rewrite relation* $\to_\mathcal{P}$ is defined as $\mathcal{C}[l\sigma] \to_\mathcal{P} \mathcal{C}[r\sigma]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\sigma \in Subst$. There the subexpression $l\sigma$ is called the redex used in that rewriting step. Notice that σ can instantiate extra variables to any expression. For any binary relation $\mathcal{R}$ we write $\mathcal{R}^*$ for the reflexive and transitive closure

[2] We use the terminology Exp (for general expressions) instead of the more usual in the field of term rewriting $Term$ (for terms), in order to highlight the syntactic (and semantic) difference with $CTerm$ (data values).

of $\mathcal{R}$. In particular that implies that we write $\xrightarrow{*}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\rightarrow_{\mathcal{P}}$, and say that $e_1 \xrightarrow{*}_{\mathcal{P}} e_2$ is a term rewriting *derivation* or *reduction* from e_1 to e_2. When presenting derivations sometimes we will underline the redexes used in each rewriting step, for the sake of readability. In the following, we will usually omit the reference to $\mathcal{P}$ or denote it by $\mathcal{P} \vdash e \rightarrow e'$ and $\mathcal{P} \vdash e \rightarrow^* e'$. We say that a TRS $\mathcal{P}$ is confluent iff $\forall e, e_1, e_2 \in Exp$ if $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$ then $\exists e_3 \in Exp$ such that $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$.

Another characterization of term rewriting is the rewriting logic of Meseguer [MM02], which comes from the field of algebraic specification. This approach is more expressive than the simple untyped framework of term rewriting, and although it has been used in many works to reason about term rewriting computations—as for example in chapter 8 of [TeR03], where rewriting logic is used to study the equivalence of term rewriting reductions—, we will stick to the simple untyped term rewriting relation defined by $\rightarrow_{\mathcal{P}}$, as no additional technical machinery will be necessary in this thesis.

A *constructor-based term rewriting system* $\mathcal{P}$ (CS), also called *program* along this work, is a particular class of term rewriting system where the rewrite rules follow the *constructor discipline*, i.e., are c-rewrite rules and so have the form $f(\bar{t}) \rightarrow r$ where $f \in FS^n$, $r \in Exp$ and $\bar{t}$ is a linear n-tuple of c-terms. As it happens with general TRS's, those CS's where extra variables are allowed are called CS's with extra variables.

During this thesis we are devoted to the study of the different semantics that can be assigned to CS's using c-terms as our notion of value, therefore we will focus on CS's from now on. As mentioned in Sect. 1.1, term rewriting is considered a standard formulation of (angelic non strict) run-time choice. The point is that the set of c-terms reachable by term rewriting from a given expression is considered to be the set of values computed for that expression under a run-time choice semantics. We will illustrate it with the following example:

Example 3.1.1. Consider the program $\mathcal{P} = \{heads(X : Y : Xs) \rightarrow (X, Y), repeat(X) \rightarrow X : repeat(X), coin \rightarrow 0, coin \rightarrow 1\}$ where $0, 1 \in CS^0$ and $\_ : \_ \in CS^2$ ($\_ : \_$ is an infix data constructor, as usual in functional languages). This program is interesting because it contains some fundamental features of lazy non-deterministic functional-logic programming: pattern matching is used to decompose an argument in the rule for *heads*; a infinite structure (a list in this case) is generated in the rule for *repeat*, this structure can be later used without necessarily incurring in non-termination thanks to lazy evaluation; finally expressions may have more than one value by using non-deterministic functions like *coin*, that may return more than one value for each configuration of its arguments (in this case there is no argument). Let us see what happens when we evaluate the expression $heads(repeat(coin))$ with term rewriting under that program:

$$\begin{aligned}&\mathcal{P} \vdash heads(\underline{repeat(coin)}) \rightarrow heads(coin : \underline{repeat(coin)})\\ &\rightarrow \underline{heads(coin : coin : repeat(coin))} \rightarrow (\underline{coin}, coin) \rightarrow (0, \underline{coin}) \rightarrow (0, 1)\end{aligned}$$

This is the behaviour expected for a run-time choice semantics, because as soon as we get an expression matching the left hand side of a program rule we are able to apply that rule, even when the values of some arguments have not been fixed yet: those values will be fixed later, during the computation, *at run-time*. In other similar derivations the values $(0, 0)$, $(1, 0)$ and $(1, 1)$ are also computed.

$$\textbf{RR}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad \textbf{DC}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\quad c \in CS^n$$

$$\textbf{B}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{OR}\ \frac{e_1 \twoheadrightarrow p_1\theta \ldots e_n \twoheadrightarrow p_n\theta \;\; r\theta \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t}\quad (f(p_1,\ldots,p_n) \to r) \in \mathcal{P},\ \theta \in CSubst_\bot$$

Figure 3.1: Rules of *CRWL*

This is why *term rewriting is considered a standard formulation for run-time choice.* As already mentioned in Sect. 1.1, run-time choice is the semantic alternative of choice for the non-deterministic specification languages Maude [CDE+07], CafeOBJ [FN97] or Elan [vdBMR02]. Although in the field of FLP run-time choice has been rarely [Ant97] thought as a valuable global alternative to call-time choice, this has not prevented the use of term rewriting as the fundamental technical tool in several foundational papers in the FLP community, in particular those devoted to the on-demand evaluation strategies [Ant92, AEH94, Esc03] used in the majority of FLP implementations, like *Toy* and *Curry*.

3.1.2 CRWL

In the ***CRWL* framework** [GHLR96, GHLR99], programs are CS's with extra variables, also called *CRWL-programs* (or simply 'programs') from now on. The original *CRWL* logic considered also the possible presence of *joinability* constraints as conditions in rules in order to give a better treatment of strict equality as built-in, which is a subject orthogonal to the aims of this work. Furthermore, due to the semantic given to equality in functional logic and thanks to the allowance of extra variables in rules, it is possible to replace conditions by the use of an *if_then* function, as has been technically proved in [SH04] for *CRWL* and in [Ant05] for term rewriting. Therefore, we consider only unconditional rules.

To deal with non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot, CTerm_\bot, Subst_\bot, CSubst_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$. The *shell* $|e|$ of an expression e represents its outer constructed part, i.e., outer and fixed *computed* part, and is defined by $|X| = X$; $|c(e_1,\ldots,e_n)| = c(|e_1|,\ldots,|e_n|)$; $|f(e_1,\ldots,e_n)| = \bot$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, meaning informally that t is (or approximates to) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice. The *CRWL*-proof calculus is presented in Fig. 3.1. Rule B (bottom) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) allow us to reduce any variable to itself, and to decompose the evaluation of a constructor-rooted expression. Finally rule OR (outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of a partial c-substitution θ) and then reduce the instantiated right-hand side. The use of $CSubst_\bot$ is essential to express call-time choice, because then only

single partial values are used for parameter passing. Notice also that by the effect of θ in OR, extra variables in the right-hand side of a rule can be replaced by any c-term, but not by any expression as in the notion of ordinary rewriting $\rightarrow_{\mathcal{P}}$.

We write $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in the *CRWL*-calculus using the program $\mathcal{P}$. Given a program $\mathcal{P}$, the *CRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$, that is, the set of its provable value approximations. In the following, we will usually omit the reference to $\mathcal{P}$. Note that these value approximations are finite by construction, as we can see just considering the definition of $CTerm_\bot \ni t ::= \bot \mid X \mid c(t_1, \ldots, t_n)$.

Example 3.1.2. Consider the program of Ex. 3.1.1 (page 21) again, the following is a *CRWL*-derivation for the statement $heads(repeat(coin)) \twoheadrightarrow (0,0)$.

$$\dfrac{\dfrac{\dfrac{}{0 \twoheadrightarrow 0}\,DC}{\mathbf{coin} \twoheadrightarrow \mathbf{0}}\,OR \quad \dfrac{\dfrac{}{0 \twoheadrightarrow 0}\,DC \quad \dfrac{\dfrac{\dfrac{}{0 \twoheadrightarrow 0}\,DC \quad \dfrac{\dfrac{}{repeat(0) \twoheadrightarrow \bot}\,B}{0 : repeat(0) \twoheadrightarrow 0 : \bot}\,DC}{repeat(0) \twoheadrightarrow 0 : \bot}\,OR}{0 : repeat(0) \twoheadrightarrow 0 : 0 : \bot}\,DC}{repeat(coin) \twoheadrightarrow 0 : 0 : \bot}\,OR \quad \dfrac{\dfrac{}{0 \twoheadrightarrow 0}\,DC \quad \dfrac{}{0 \twoheadrightarrow 0}\,DC}{(0,0) \twoheadrightarrow (0,0)}\,DC}{heads(repeat(coin)) \twoheadrightarrow (0,0)}\,OR$$

This is the behaviour expected for a call-time choice semantics, a value for each function argument is computed on parameter passing, i.e, *at call-time*. If each argument value matches its corresponding argument in a given program rule then the corresponding matching substitution is applied to the right hand side of that program rule to continue the computation. Note that the value 0 for the subexpression *coin* of the reduced expression $heads(repeat(coin))$ is already fixed when evaluating the argument of *repeat*, as a consequence of evaluating the argument of *heads*. Thus *coin* is evaluated only once and no 1 is present in the proof, so the expressions $(0,1)$ and $(1,0)$ are not valid values for $heads(repeat(coin))$, contrary to what happens under run-time choice (see Ex. 3.1.1). We can try all the possible *CRWL*-derivations starting from $heads(repeat(coin))$ to conclude that $[\![heads(repeat(coin))]\!] = \{(0,0), (1,1), (\bot,0), (\bot,1), (0,\bot), (1,\bot), (\bot,\bot), \bot\}$.

The fact of working with finite approximations makes it unnecessary to use a background of cpo's and powerdomains. This is very convenient, specially as it is known that an approach based on semantic domains with infinite (limit) elements and using fixpoint techniques has technical limitations when dealing with non-deterministic languages [Bou81, Nys96]. Then *CRWL*-denotations are possibly infinite sets of finite partial values, that intuitively correspond to finite approximations to possibly infinite values. But the infinite values themselves are not technically used in the *CRWL* framework.

Example 3.1.3. We can see an example of this by considering again the program of Ex. 3.1.2 and the expression $repeat(coin)$, for which it is easy to see that its *CRWL*-denotation is the infinite set

$$\{\bot, \bot{:}\bot, \bot{:}\bot{:}\bot, \ldots, 0:\bot, 0:\bot{:}\bot, \bot{:}\,0:\bot, 0:0:\bot, \ldots, 1:\bot, 1:\bot{:}\bot, \bot{:}\,1:\bot, 1:1:\bot, \ldots\}$$

containing all the finite approximations to the 'limit values' $0:0:0:\ldots$ and $1:1:1:\ldots$

We stress the fact that the *CRWL-calculus* is not an operational mechanism for executing programs, but a way of describing the meaning of programs, and in particular

is considered a standard formulation for call-time choice [Han07]. At the operational level the *CRWL* framework comes with various lazy narrowing-based goal-solving calculi [GHLR99, LRV04, Vad03]. Finally, several extensions of the *CRWL* framework have been developed through the years, adding higher order capabilities [GHR97], type systems and algebraic datatypes [AR01, GHR01], constraint systems [Rod01, LRV07, EFS+09], constructive failure [SH04], qualified functional-logic programming [CRR09], ...

3.1.3 *FLC*: Flat Curry Semantics

The operational semantics of [AHH+05] and its variants [BH07, BHH04] constitute an important family of semantic descriptions in the FLP community. Through this work we will refer to these semantics as the ***FLC* framework**, for its proximity to *Flat Curry* [HP99]. Here we will outline some of the main characteristics of the original version of *FLC*, as it was first presented in [AHH+05].

The language *FLC* considered in [AHH+05] is a convenient low-level format to which programs used in modern FLP implementations like *Toy* or *Curry* can be transformed (not in a unique manner). This transformation embeds important aspects of the operational procedures of FLP languages, like the use of definitional trees [Ant92, LLR93]. The syntax of Flat Curry programs is given in Fig. 3.2. Notice that each function symbol f has exactly one definition rule $f(x_1, \ldots, x_n) = e$ with distinct variables $x_1, \ldots, x_n$ as formal parameters. All non-determinism is expressed by the use of *or* choices in right-hand sides and moreover all pattern matching has been moved to right-hand sides by means of nesting of *(f)case* expressions. The language distinguishes between rigid *case* expressions, which perform pattern matching but not narrowing, and flexible *fcase* expressions, which also perform narrowing. This corresponds to the distinction between rigid and flexible functions allowed in the *Curry* language [Han06]. Finally *let* bindings are a convenient way to express sharing.

Programs: $P ::= D_1 \ldots D_m$	
Function definitions: $D ::= f(x_1, \ldots, x_n) = e$	
Expressions	
$e ::= x$	(variable)
$\mid\ c(e_1, \ldots, e_n)$	(constructor call)
$\mid\ f(e_1, \ldots, e_n)$	(function call)
$\mid\ case\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$	(rigid case)
$\mid\ fcase\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$	(flexible case)
$\mid\ e_1\ or\ e_2$	(disjunction)
$\mid\ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$	(let binding)
Patterns: $p ::= c(x_1, \ldots, x_n)$	

Figure 3.2: Syntax for FLC programs

An additional *normalization step* over programs is assumed in [AHH+05]. In normalized expressions each constructor or function symbol appears applied only to distinct variables. This can be achieved by repeatedly introducing a *let* binding for each non variable argument: for example $f(g)$ is transformed into $let\ x = g\ in\ f(x)$. The normalization of e is written as e^*. Notice that any *CRWL*-expression e is also a *FLC*-expression, and therefore we can speak of its normalization e^*.

In [AHH+05] two operational semantics for the *FLC* language are given: a natural (*big-step*) semantics in the style of Launchbury's operational semantics for lazy evaluation (with sharing) of functional programs [Lau93], and a small step semantics. In Fig. 3.3 we may find the natural semantics, we refer the reader to [AHH+05] for the small-step version. This natural semantics uses configurations of the shape $\Gamma : e$ where e is a normalized *FLC* expression and Γ is a *heap*, i.e., a finite map from variables to normalized *FLC* expressions. The empty heap is $[]$, $\Gamma[x \mapsto e]$ denotes a heap Γ' such that $\Gamma'[x] = e$ and $\Gamma'[y] = \Gamma[y]$ for every $y \neq x$, and $\Gamma[x]$ denotes the expression associated to x in Γ. Thus this semantics consists of a set of rules defining a relation $\Gamma : e \Downarrow \Delta : v$, indicating that one of the possible evaluations of e ends up with the head normal form—variable or constructor rooted *FLC* expression—v. The initial configuration to evaluate an *FLC* expression e is $[] : e^*$.

(VarCons) $\Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t$ $\quad$ t constructor-rooted

(VarExp) $$\frac{\Gamma[x \mapsto e] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v}$$ e not constructor-rooted, $e \neq x$

(Val) $\Gamma : v \Downarrow \Gamma : v$ $\quad$ v constructor-rooted or variable with $\Gamma[v] = v$

(Fun) $$\frac{\Gamma : e\rho \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v}$$ $f(\overline{y_n}) = e \ \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

(Let) $$\frac{\Gamma[\overline{y_k \mapsto e_k\rho}] : e \Downarrow \Delta : v}{\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Delta : v}$$ $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh variables

(Or) $$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ or\ e_2 \Downarrow \Delta : v}$$ $i \in \{1, 2\}$

(Select) $$\frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : e_i\rho \Downarrow \Theta : v}{\Gamma : (f)case\ e\ of\ \{\overline{p_k \mapsto e_k}\} \Downarrow \Theta : v}$$ $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$

(Guess) $$\frac{\Gamma : e \Downarrow \Delta : x \qquad \Delta[x \mapsto p_i\rho, \overline{y_n \mapsto y_n}] : e_i\rho \Downarrow \Theta : v}{\Gamma : fcase\ e\ of\ \{\overline{p_k \mapsto e_k}\}\ in\ e \Downarrow \Theta : v}$$ where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables

Figure 3.3: Natural Semantics for *FLC*

Example 3.1.4. We can translate the program of Ex. 3.1.1 (page 21) to normalized *FLC* format as follows:

$$\begin{array}{lcl} heads(x) & = & case\ x\ of\ \{x_1 : y_s \to case\ y_s\ of\ \{x_2 : xs \to (x_1, x_2)\}\} \\ repeat(x) & = & let\ y = repeat(x)\ in\ x : y \\ coin & = & 0\ or\ 1 \end{array}$$

Now, in order to evaluate the expression $e \equiv heads(repeat(coin))$, we must first normalize it getting the expression $e^* \equiv let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l)$, and then apply the rules of Fig. 3.3 to the initial configuration $[] : e^*$ (some steps have been omitted

for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3 : c_1 : y_1 \Downarrow \Gamma_3 : c_1 : y_1}\,Val}{\Gamma_2 : let\ y = repeat(c_1)\ in\ c_1 : y \Downarrow \Gamma_3 : c_1 : y_1}\,Let}{\Gamma_2 : repeat(c_1) \Downarrow \Gamma_3 : c_1 : y_1}\,Fun}{\Gamma_1 : let\ c = coin\ in\ repeat(c) \Downarrow \Gamma_3 : c_1 : y_1}\,Let}{\Gamma_1 : l_1 \Downarrow \Gamma_4 : c_1 : y_1}\,VExp \quad \dfrac{\dfrac{\dots}{\Gamma_4 : y_1 \Downarrow \Delta : c_1 : y_2}\,VExp \quad \dfrac{}{\Delta : (c_1, c_1) \Downarrow \Delta : (c_1, c_1)}\,Val}{\Gamma_4 : case\ y_1\ of\ \{x_2 : x_s \to (c_1, x_2)\} \Downarrow \Delta : (c_1, c_1)}\,Select}{\Gamma_1 : case\ l_1\ of\ \{x_1 : y_s \to case\ y_s\ of\ \{x_2 : x_s \to (x_1, x_2)\}\} \Downarrow \Delta : (c_1, c_1)}\,Select$$

$$\dfrac{\Gamma_1 : heads(l_1) \Downarrow \Delta : (c_1, c_1)}{[] : let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l) \Downarrow \Delta : (c_1, c_1)}\,Let}\,Fun$$

where

- $\Delta \equiv [l_1 \mapsto c_1 : y_1, c_1 \mapsto coin, y_1 \mapsto c_1 : y_2, y_2 \mapsto repeat(c_1)]$
- $\Gamma_1 \equiv [l_1 \mapsto let\ c = coin\ in\ repeat(c)]$
- $\Gamma_2 \equiv [l_1 \mapsto let\ c = coin\ in\ repeat(c), c_1 \mapsto coin]$
- $\Gamma_3 \equiv [l_1 \mapsto let\ c = coin\ in\ repeat(c), c_1 \mapsto coin, y_1 \mapsto repeat(c_1)]$
- $\Gamma_4 \equiv [l_1 \mapsto c_1 : y_1, c_1 \mapsto coin, y_1 \mapsto repeat(c_1)]$

Several characteristics of *FLC* are revealed in this derivation. First of all, as it works with normalized expressions only, we cannot expect to prove something like $[] : e^* \Downarrow \Theta : (0, 0)$, because $(0, 0)$ is not a normalized expression. Neither we can expect to prove $[] : e^* \Downarrow \Theta : (x, x)$ for a heap Θ where $\Theta[x] = 0$, because by rule (Val) this semantics stops evaluation as soon as we reach a head normal form. Nevertheless the resulting configuration shows a behaviour corresponding to a call-time choice semantics too, because there is only one appearance of *coin* in the resulting heap, and the rule (VarExp) ensures that it would be evaluated only once. That evaluation could be forced if the original expression $heads(repeat(coin))$ was put in a context demanding the complete evaluation of its arguments, for example $f([])$ where f is defined by $f((x, y)) = case\ x\ of\ \{0 \to case\ y\ of\ \{0 \to true; 1 \to false\}; 1 \to case\ y\ of\ \{0 \to false; 1 \to true\}\}$. It is easy to see that $f(heads(repeat(coin)))$ could only be reduced to *true* in this semantics.

The *FLC* framework provides an operational semantics designed to reason at an abstraction level close to current FLC implementations. This is reflected in the way some of the most important features of modern FLP languages, namely lazy evaluation, call-time choice semantics and narrowing, are reflected in the calculus:

- *Lazy evaluation*: source FLP programs are transformed into *FLC* programs in which the use of the *case* construction makes explicit the demandness of evaluation that will be performed by pattern matching and narrowing. Several alternative transformations can be conceived by using the different demandness analysis performed by concrete strategies [AEH94, Esc03], thus encoding a particular on-demand strategy in the *FLC* program. Each strategy has its own optimality properties, as well as a cost to implement it, because it may involve a different exploration of the search space. None of these features is reflected in the *FLC* framework because a transformation is assumed a priori, nevertheless *FLC* can still be useful to reason about the matching process itself.
  Moreover the evaluation of expressions stops as soon as a head normal form is reached, again something very close to the behaviour of implementations, and hence useful for reasoning at that low abstraction level.

- *Call-time choice* is obtained by a combination of factors. First of all the normalization process ensures that every subexpression is "shared", i.e., bound in a *let* construction. These bindings are refreshed and introduced in the heap by means of the (Let) rule, when its evaluation is demanded. Then rules (VarExp) and (Val) ensure that any binding is evaluated only once, thus getting a call-time choice behaviour.

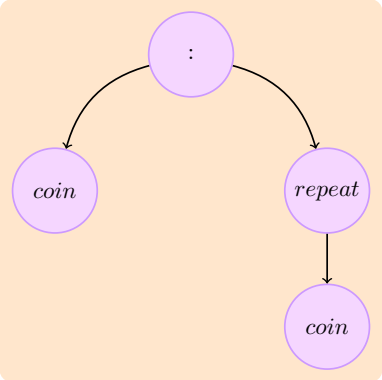
- *Narrowing* is expressed through the (Guess) rule, which non-deterministically chooses a pattern from a flexible *case* distinction to instantiate a free variable that has been computed for the expression used to resolve that *case* choice.

Therefore the resulting operational semantics models each of these features. Although it works at a pretty low abstraction level, the only computational structure introduced is the heap, thus it still remains at a higher level of abstraction than operational semantics for particular abstract machines, and proofs made with this semantics can be applied to several concrete implementations. Finally, the use of a heap makes it specially suitable to reason about space behaviour of programs, a characteristic inherited from [Lau93].

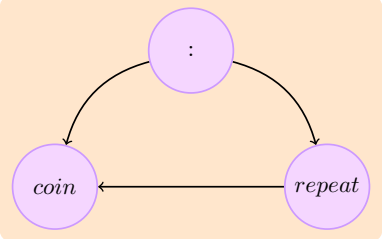
3.1.4 Term Graph Rewriting

Term graph rewriting is a formalism that can be considered an extension of term rewriting in some way, in which expressions—terms in the traditional nomenclature of the field of term rewriting, elements of Exp in this thesis—are represented as graphs thus allowing the sharing of common subexpressions.

Example 3.1.5. Consider again the program of Ex. 3.1.1 (page 21), and the expression $coin : repeat(coin)$. It is natural to represent this expression as a tree, which by the way describes its structure.



But we may also represent the same term with the following *term graph*.



This representation has several advantages. First of all it saves space, because we need just one instance of the *coin* symbol, which is referred two times, instead of having two different copies of *coin*. Furthermore, there is a potential saving of time because as soon as we evaluate *coin* its value will be available from any arrow pointing to the node for *coin*, while in the tree representation the different instances of *coin* cannot *share* their evaluations and values.

Term graph rewriting has been used for a long time to improve the efficiency of implementations of term rewriting [CDE+07] or functional languages [Pla95, PJ87]. In [BEG+87] Barendregt et. al. proved that term graph rewriting is a sound and complete implementation of term rewriting for the class of orthogonal TRS's, therefore the use of term graph rewriting in a deterministic setting like those of classical functional programming is pretty natural. On the other hand for general implementations of term rewriting some additional manipulation of the sharing structure of graphs is needed in order to achieve completeness.

Besides Barendregt's groundbreaking work [BEG+87], we can find several variants and formulations of term graph rewriting in the bibliography.

- In [Plu99] a formulation of acyclic term graph rewriting—i.e., where the graphs used to represent terms have no cycles—based on hypergraphs is proposed. There the notion of graph morphism in introduced, and used to define collapsing and copying of graphs, as well as the notion of term graph rewriting step. This mainly follows the algorithmic approach of [BEG+87], where a rewriting step is decomposed, after a matching redex is found, in three consecutive phases: in the build phase the graph corresponding to the instance of the right hand side of the program rule used is added to the graph being rewritten; in the redirection phase pointers in the graph are redirected to perform parameter passing; finally in the garbage collection phase nodes not reachable from the root of the graph are eliminated. The adequacy of term graph rewriting as a mechanism to simulate term rewriting is also studied, and the conclusion is that term graph rewriting is always sound wrt. term rewriting, but collapsing and copying steps (see Fig. 3.4) are sometimes unavoidable in order to get completeness for any TRS. Finally the termination and confluence problems for term graph rewriting are tackled, and a formulation of narrowing for term graphs is also proposed.
- Chapter 13 of [TeR03] deals with a variant of term graph rewriting in which cycles

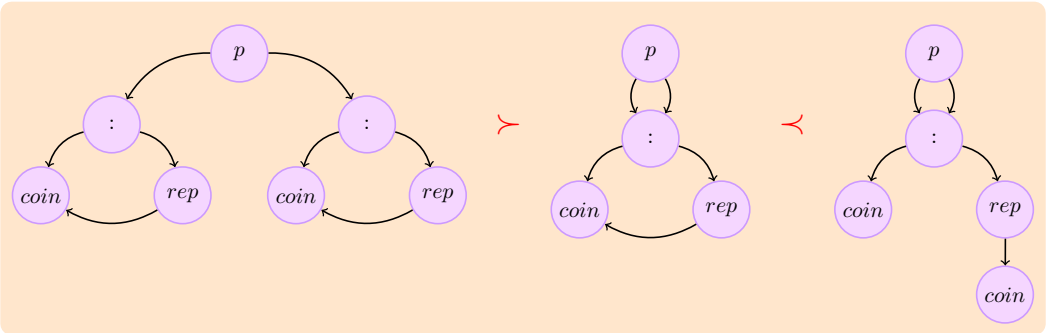


Figure 3.4: Collapsing ($\succ$) and copying ($\prec$)

are permitted. This is called "vertical sharing", which is different to the usual sharing of acyclic term graphs, called "horizontal sharing" there. Three different characterizations of term graphs are presented, the first one follows the style of [BEG+87] so a graph is a set of nodes with a distinguished root node, a function to assign elements of Σ to the nodes and a function to attach other nodes as arguments of one node. These functions must respect the arity of symbols, and variables are represented by "empty nodes". The second representation, called "equational representation", is very similar to the heap approach of *FLC* and the linear notation of [BEG+87]: a graph specification is basicly a mapping from node variables to expressions together with a distinguished root node variable. Normalization—in the sense of *FLC*—is not needed but sometimes used, and normalized graph specifications are told to be in "flattened form". The third alternative representation of cyclic graphs is based on μ-terms, which an extension of expressions where a construction $\mu x.e$ is intuitively used to indicate that x is a pointer to the root of e that can also be used in its subexpressions. Although not every graph can be written as a μ-term, they form an interesting class of graph representations.

In each of these representations the notion of homomorphism ordering on graphs, which corresponds to the notion of collapsing of graphs, as well as other relations between graphs, are defined. Term graph rewriting is also formulated for the equational representation.

- In [Ohl02] another formulation of acyclic term graph rewriting is presented, based on the framework of marked terms. In this approach the syntax of terms in extended by adding a mark, usually a natural number, to each symbol (of the signature or variable) that is part of an expression. The point is that two marked subexpressions which are syntactically equal—which implies that they have the same marks—represent two pointers to a single shared expression. Conversely, independent apparitions of the same term are distinguished by using different marks. The resulting framework is expressive enough to represent both term rewriting and term graph rewriting, for which the notions of collapsing, copying and term graph rewriting step are defined.

- In Chapter 5 of [PvE93] we may find another formulation of cyclic term graph rewriting, more oriented to the use of term graph rewriting in the implementation of functional languages. Two syntactic ways of specifying graphs are proposed, the canonical form and the shorthand form. The fist one is very similar to the flattened form of the equational representation of [TeR03], and is used to reason about the semantics of term graph rewriting, in particular to define the notion of term graph rewriting step. A crucial ingredient there is the specification of node redirections to define the term graph rewriting rules. Again this goes back to Barendregt's formulation [BEG+87], where graph rewriting rules are triples which first component is a (often many rooted) graph specifying the shape of the left and right hand sides of the rule, and the second and third components are two nodes in that graph which specify the redirection to be performed on rewriting. On the other hand the shorthand form is a convenient abbreviated format in which some information about the graph structure is not explicitly specified but can be deduced from the notation. It is used for specifying programs, as it allows term graph rewriting rules written in this format to be closer to regular term rewriting rules, which is useful for rules where there is no interesting sharing manipulation.

- In [CMR+97, EHK+97] we may find two algebraic approaches to graph transformation, the double-pushout approach (DPO) and the single-pushout approach (SPO). There graphs are considered as special kinds of algebras, and categorical notions are used for the definitions of graph rewriting rule, graph matching and graph rewriting step. The difference between these approaches is the way rewriting steps are defined, either with two pushouts in the category of graphs and total graph morphisms, in the case of DPO, or with a single pushout in the category of graphs and partial graph morphisms, in the case of SPO. The main advantage of both is that techniques from the field of category theory can then be used for reasoning about graph transformations.

- In [Cou90] mathematical logic, category theory and universal algebra are used to manipulate graphs. There, by graph rewriting the authors refer not only to graph rewriting systems, but also to context-free graph grammars, and descriptions of infinite graphs and sets of infinite graphs by the use of rewriting rules and systems of equations. The aforementioned mathematical tools are used in several ways. Graphs are considered as logical structures so logical formulas are used to express their properties: then a logical formula is used to characterize the set of graphs satisfying the corresponding property. Category theory is used both for specifying graph rewriting rules and for defining the initial solution of a system of graph equations. Finally every finite graph can be represented by a finite algebraic expression called graph expression, hence graph rewriting systems can be considered as term rewriting systems that rewrite these graph expressions.

- The approach of [EJ97, EJ98] is mostly focussed on the use of term graph rewriting and narrowing as the operational semantics for FLP. It manipulates *admissible graphs*, which are a restriction of cyclic graphs where no function symbol can appear in a cycle. As usual in FLP and functional languages in general, the constructor discipline is used so programs are described by *constructor-based graph rewriting systems* (*cGRS's*)—also called admissible graph rewriting systems (AGRS) in [EJ98]—, which are a natural extension to graphs of the class of orthogonal constructor-based TRS's [BN98, TeR03]. In this setting not only term graph rewriting is defined, but also its extension to narrowing over admissible graphs, for which soundness and completeness wrt. term graph rewriting are proved. Besides, the on-demand evaluation strategies of [Ant92, AEH94] for term rewriting and narrowing are adapted to this framework.

  The classical notation for graphs of [BEG+87] is used, both that which describes graphs as a tuple $< nodes, labelling\ function, successor\ function, root\ nodes >$— also followed in the first formulation of [TeR03]— and the linear notation—pretty similar to the canonical form of [PvE93] and the flattened form of the equational representation of [TeR03].

- Finally there are several formulations of the semantic of programming languages that, although are not meant to manage term graphs, indeed perform an implicit manipulation of term graphs. We have already mentioned that it is the case for *FLC*, and thus for its "parent" semantics, the operational semantics for lazy evaluation of an extended λ-calculus proposed by Launchbury [Lau93]. In both semantics we may consider that the heap represents a term graph that evolves during the evaluation process—to be more precise it would represent a set of graphs, as no garbage

collection is explicitly performed by the semantics.

Something similar happens in the call-by-need lambda calculi of [AFM+95, AF97, MOW98], where the *let* construction corresponds to a pointer in an acyclic graph. And we can even find a similar situation in the goal solving calculus *CLNC* (Constructor-based Lazy Narrowing Calculus) for *CRWL* and its variants [GHLR99, GHR97, GHR01], in which produced variables of goals again play the role of pointers to the nodes of an implicit graph.

As we said before, term graph rewriting is often used to improve the implementations of functional languages. In these deterministic settings the sharing of values between the different copies of a function argument made by evaluation does not change the semantics of systems, but just improves the performance both in space—as it allows a more compact representation of expressions—and in time—because instead of making several copies of a costly (in evaluation time) expression we can replace it by several pointers to a single graph node containing that expression, and evaluate it only once instead of one time for each copy.
Nevertheless the use of sharing can change the semantics in a non-deterministic setting, as we will see considering Ex. 3.1.1 (page 21) again. If we evaluate the term graph corresponding to the expression $heads(repeat(coin))$ by just performing term graph rewriting steps, with no copy or collapsing step involved, then the two copies of X made in the rule for *repeat* will point to the same node in the graph, thus sharing their values. Henceforth the term graph corresponding to $heads(repeat(coin))$ could not be reduced neither to $(0, 1)$ nor to $(1, 0)$: *in a non-deterministic setting the operational notion of sharing leads naturally to an implementation of the semantic notion of call-time choice.*

This is why term graph rewriting has been used in many works in the field of FLP to reason about programs under a call-time choice semantics. *The formulation of [EJ97, EJ98]* is the most popular in the FLP community, and have been used in many works, for example [ABC07, ABC06]. Although never formally proved, it is usually considered that this formulation of term graph rewriting *specifies the same behaviour as CRWL and FLC, i.e., a call-time choice semantics.*

3.1.5 Non-determinism and Functional Programming

There have been an interest in non-determinism in the functional programming community from a long time ago. In [HO90] we can find an interesting look at some ways of adding non-determinism to a functional language. There, some classical proposals are reviewed, like McCarthy's *amb* bottom avoiding choice operator [McC63], Henderson's *merge* operator [Hen80], Stoye's scheme for a functional operating system where non-determinism is placed in the communications of process instead of in the programs [Sto84] and Holström's PFL [Sör83], a combination of ML [Wik87] with CCS [Mil82]. In [HO90] Hughes and O'Donnell also propose a new approach based on the introduction of a set datatype so non-deterministic functions are those whose result is a set.

Another interesting reference is [SS92]. In that work, after a discussion over different dimensions of non-determinism (weak or strong, bounded or unbounded, angelic demonic or erratic, restrained or unrestrained), a very simple functional language and its denotational semantics are presented. Then a non-deterministic alternative operator is added to the language and twelve different possible semantics are considered by extending the original

denotational semantics and making choices between singular or plural semantics, strict or non strict functions, and angelic, demonic or erratic choice.

We can find several other works where functional programming is extended with non-deterministic features. In [HM95] a natural semantics for an extension of λ-calculus extended with McCarthy's *amb* in a call-by-need context was developed, resulting in a singular semantics for *amb*. In [LM99] an operational theory for fair non-determinism in a higher order call-by-name functional programming extended with *amb* is developed. In [KSS98] a non-deterministic call-by-need λ-calculus with a *choice* operator and *let* syntax to model sharing was presented; this work was latter extended in [SSH00], where *case*, *constructors* and *letrec* where added to the framework.

Other works in the functional community are devoted to the simulation of non-determinism in a pure functional language, without the need of extending it with additional constructions. A classical text is [Wad85], where Haskell's [PJ03] list datatype is used as the basis to represent non-deterministic values. This technique has been later standardized through Haskell's `MonadPlus` class [Wik10], for which the list datatype is its more simple instance [Has10a]. Several works were later developed to devise different representations of non-determinism in Haskell with an improved performance [Hin00, KSFS05, FKS09]. An interesting application of these ideas can be found in [NAR07], where a combination of the non-deterministic monad of [Hin00] and a state monad [Has10b] were used to implement not only non-determinism but also the narrowing procedure on which FLP systems are based, in order to develop a relational domain-specific language for describing and analyzing digital circuits. This subject of representing non-determinism in deterministic functional languages has been also treated for other functional languages too, as it is the case for Scheme [ABB+98] in [FBK05].

Non-determinism is also present in functional programming through the use of the primitives for concurrent or parallel programming available in some functional programming languages. This is the case for languages like Erlang [Arm07] or Scala [Pol09], which are pragmatic languages designed to develop concurrent systems, and less concerned with the elaboration of the pure functional programming paradigm. Nevertheless there are also extensions of some paradigmatical functional languages to support concurrency or parallel evaluation. Examples of that are Concurrent ML [Rep91], Eden [LOP05], Glasgow Parallel Haskell [AZTML08], NESL [Ble93], Manticore [FRR+07], Data Parallel Haskell [PJ08], the `Control.Concurrent.STM` library of Haskell [HMJH08] or just simply the basic concurrency primitives of Haskell [LMJT07]. There are also several works formalizing the semantics of these extensions [HH04], but as we said in Sect. 1.1 this thesis is devoted to the study of the use of non-determinism as a language expressive feature, therefore the analysis of the consequences of introducing concurrent or parallel evaluation in a functional language is out of the scope of this work.

3.2 Call-time Choice

"We usually think that if something is not one, it is more than one; if it is not singular, it is plural. But in actual experience, our life is not only plural, but also singular. Each one of us is both dependent and independent."

Shunryu Suzuki — Zen Mind, Beginner's Mind - 1970

3.2.1 CRWL vs. FLC

In this section we will summarize the equivalence results for *CRWL* and *FLC* obtained in [LRS07a] (Sect. 7.2.2, page 140). Our goal is to relate both approaches in a technically precise manner, so in this way some known or future results obtained for one of them could be applied to the other.

There are important differences between *FLC* and $CRWL_{FLC}$ that complicates the task of relating them. The heaps used in *FLC* for storing variable bindings have not any (explicit) correspondence in *CRWL*. Another important difference is that the first one obtains *head normal forms* for expressions, while the second is able to obtain any value of the denotation of an expression (in particular a normal form if it exists).

Differences do not end here. There are still two important points that enforce us to take some decisions: (1) *FLC* performs narrowing while *CRWL* is a pure rewriting relation. We have addressed this inconvenience by considering only the rewriting fragment of *FLC*. Narrowing acts in *FLC* either due to the presence of logical variables in expressions to evaluate or because of the use of extra variables in program rules (those not appearing in left-hand sides). So we can isolate the rewriting fragment by excluding this kind of variables throughout this work. Therefore, we assume that programs do not have extra variables and that expressions to be reduced are ground. (2) The other difference stems from the fact that *FLC* allows recursive *let* constructions. Since there is not a well established consensus about the semantics of such constructions in a non-deterministic context, and furthermore they cannot be introduced in the transformation of *CRWL*-programs, we exclude recursive *let*'s from the language in this work. In absence of recursive *let*'s it is not difficult to see that a *let* with multiple variable bindings may be expressed as a sequence of nested *let*'s, each with a unique binding. For simplicity and without loss of generality we will consider only this kind of *let*'s. We assume from now on that programs and expressions fulfil the conditions imposed in (1) and (2).

Working Plan

The first thing we have to do in order to establish the relation between *CRWL* and *FLC* is adapting *CRWL* to the syntax of *FLC*. For this purpose in Fig. 3.5 we introduce the rewriting logic $CRWL_{FLC}$ as a variant of *CRWL* with specific rules for managing *let*, *or* and *case* expressions. The first three rules, **(B)**, **(RR)** and **(DC)**, are directly incorporated from *CRWL*. Rules **(Case)**, **(Or)** and **(Let)** have also a clear reading. Finally, rule **(Red)** is a simplified version of the corresponding rule in *CRWL*, as now we can guarantee that any function call in a derivation only uses c-terms as arguments (this is easy to check assuming that we start from a normalized expression and taking into account that the rules of the calculus only apply c-substitutions).

$$\textbf{(B)}\ \frac{}{e \to \bot} \qquad \textbf{(RR)}\ \frac{}{x \to x} \qquad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{t_1 \to t_1' \ \ldots\ t_n \to t_n'}{c(t_1,\ldots,t_n) \to c(t_1',\ldots,t_n')} \qquad c \in CS^n,\ t_i, t_i' \in CTerm_\bot$$

$$\textbf{(Red)}\ \frac{e\theta \to t}{f(\overline{t}) \to t} \qquad (f(\overline{y}) = e) \in \mathcal{P},\ \theta = [\overline{y/t}]$$

$$\textbf{(Case)}\ \frac{e \to c(\overline{t}) \qquad e_i\theta \to t}{case\ e\ of\ \{\overline{p_i \to e_i}\} \to t} \qquad \begin{array}{l} p_i = c(\overline{x}) \text{ for some } i \\ \theta = [\overline{x/t}] \end{array}$$

$$\textbf{(Or)}\ \frac{e_i \to t}{e_1\ or\ e_2 \to t} \qquad \text{for some } i \in \{1,2\}$$

$$\textbf{(Let)}\ \frac{e' \to t' \quad e[x/t'] \to t}{let\ \{x = e'\}\ in\ e \to t}$$

Figure 3.5: Rules of $CRWL_{FLC}$ [LRS07a]

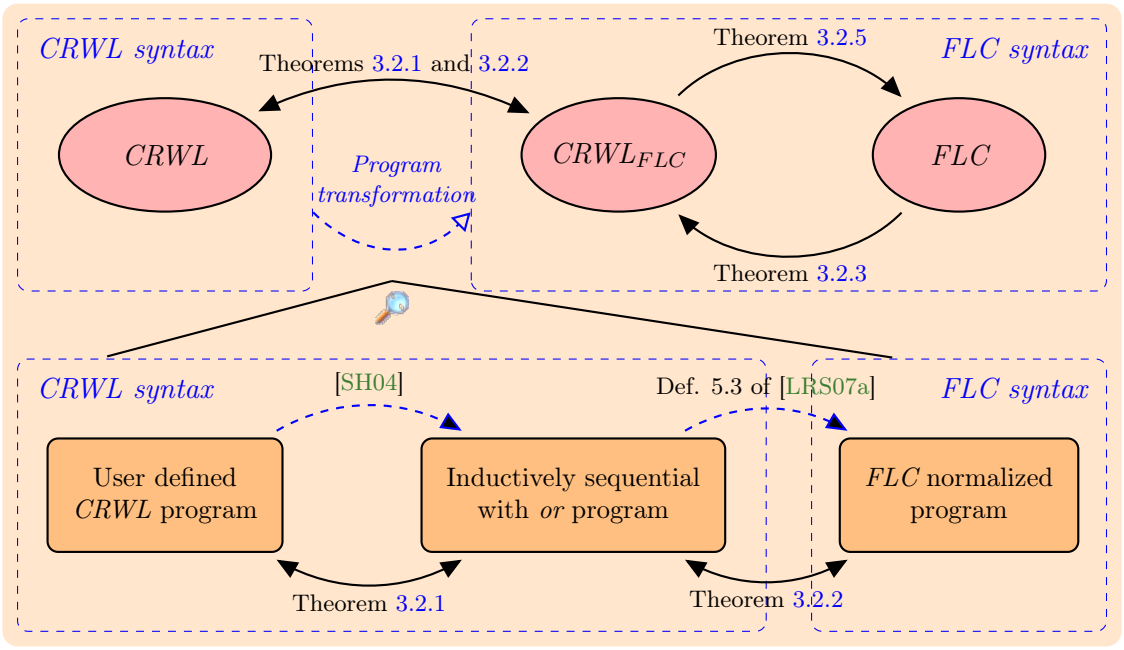


Figure 3.6: *CRWL* vs. *FLC*: Proof's plan [LRS07a]

The relation between *CRWL* and *FLC* is established through that intermediate logic. The working plan is sketched in Fig. 3.6. Given a pair program/expression in *CRWL* we transform them into *FLC*-syntax and study the semantic equivalence of both versions of *CRWL* (Theorems 3.2.1 and 3.2.2). Then we focus on the equivalence of *FLC* with respect to $CRWL_{FLC}$ in a common syntax context (Theorems 3.2.3 and 3.2.5). *FLC* and *CRWL* are very different frameworks from the syntactical and the semantical points of view. The advantage of splitting the problem is that on one hand both versions of *CRWL* are very

close from the point of view of semantics; on the other hand $CRWL_{FLC}$ and *FLC* share the same syntax.

Relation between $CRWL_{FLC}$ and *CRWL*

Our equivalence result for $CRWL_{FLC}$ and *CRWL* is based on a program transformation from *CRWL* syntax to *FLC* syntax. A similar translation is assumed but not made explicit in [AHH+05]. As seen in the zoomed part of Fig 3.6, for technical convenience we split the transformation into two parts: first, and still within *CRWL*-syntax, we transform P into another program P' which is *inductively sequential* ([Ant92, Han07]), except for a function *or* defined by the two rules X *or* $Y = X$ and X *or* $Y = Y$. The function *or* concentrates all the non-sequentiality (hence, all the indeterminism) of functions in right-hand sides, so we speak of inductively sequential with *or* (IS_{or}) programs. Such kind of transformations are well-known in functional logic programming. In the *CRWL* setting, a particular transformation has been proposed in [SH04], where it is proved the following result:

Theorem 3.2.1 ([SH04]). *Let P be a CRWL-program and $e \in Exp_\perp$ a CRWL-expression. Then $[\![e]\!]^P_{CRWL} = [\![e]\!]^{P'}_{CRWL}$ where P' is the IS_{or} transformed program of P.*

Now, to transform IS_{or} programs into normalized *FLC*-syntax we simply mimic the inductive structure of function definitions by means of (possibly nested) *case* expressions. This transformation is formulated in detail in [LRS07a] (Sect. 7.2.2, page 140), where we refer the reader to for details. However here we still present the most important property of that transformation, its correctness, which is stated in the following result.

Theorem 3.2.2 ([LRS07a] Th. 5.4). *Let P be an IS_{or} CRWL-program, $\hat{P}$ its FLC-transformation, $e \in Exp_\perp$ a CRWL-expression, and e^* its FLC-normalization. Then*

$$[\![e]\!]^P_{CRWL} = [\![e^*]\!]^{\hat{P}}_{CRWL_{FLC}}$$

Relation between *CRWL* and *FLC*

We need to express pairs heap/expression of the *FLC* formalism as *CRWL*-expressions in order to relate computations with respect to both approaches. Notice that as recursive bindings are not allowed in heaps it is always possible to order the heap $\Gamma = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ in such a way that e_i does not depend on any x_j with $j >= i$. Then it makes sense to obtain a *CRWL*-expression from a pair heap/expression as:

$$ligs([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n], e) =_{def} let\ \{x_1 = e_1\}\ in \ldots let\ \{x_n = e_n\}\ in\ e$$

With this transformation we can define:

Definition 3.2.1 ($CRWL_{FLC}$-denotation of a pair heap/expression, [LRS07a] Def. 6.1). Given an *FLC*-program $\mathcal{P}$ and a pair (Γ, e), where Γ is a valid heap and e is a *FLC*-expression, we define the denotation of the pair with respect to $CRWL_{FLC}$ as

$$[\![\Gamma, e]\!]^{\mathcal{P}}_{CRWL_{FLC}} = [\![ligs(\Gamma, e)]\!]^{\mathcal{P}}_{CRWL_{FLC}}$$

We will usually omit the reference to the program $\mathcal{P}$ and the calculus $CRWL_{FLC}$ when they are clear by the context, and write simply $[\![\Gamma, e]\!]$. Notice that $ligs([], e) = e$ and therefore $[\![[], e]\!] = [\![e]\!]$, for any e. The notion of shell, which was introduced in *CRWL* (see Sect. 3.1.2, page 22), is also adapted in Fig. 3.7 to *FLC*-expressions, and as usual it is a partial c-term which represents the constructed part of the expression.

$\lvert x\rvert$	$=$	x	$\lvert e_1\ or\ e_2\rvert$	$=$	$\bot$
$\lvert c(e_1,\ldots,e_n)\rvert$	$=$	$c(\lvert e_1\rvert,\ldots,\lvert e_n\rvert)$, if $c \in DC$	$\lvert case\ e\ of\ \{\overline{p_k \to e_k}\}\rvert$	$=$	$\bot$
$\lvert f(e_1,\ldots,e_n)\rvert$	$=$	$\bot$, if $f \in FS$	$\lvert let\ x = e_1\ in\ e_2\rvert$	$=$	$\lvert e_2\rvert[x/\lvert e_1\rvert]$

Figure 3.7: Shell of a FLC expression [LRS07a]

Completeness of *CRWL* wrt. *FLC*
The following theorem is our main completeness result of *CRWL* wrt. *FLC* and shows that any *FLC*-derivation for a pair heap-expression is captured by a $CRWL_{FLC}$-derivation of the corresponding $CRWL_{FLC}$-expression.

Theorem 3.2.3 ([LRS07a] Th. 6.2)**.** *If* $\Gamma : e \Downarrow \Delta : v$, *then* $[\![\Delta, v]\!] \subseteq [\![\Gamma, e]\!]$.

In order to prove it, we split the completeness Th. 3.2.3 into two properties: *(H)* shows what happens to heaps under a *FLC*-derivation, while *(R)* relates the results of the computation.

Theorem 3.2.4 ([LRS07a] Th. 6.4)**.** *If* $\Gamma : e \Downarrow \Delta : v$, *then:*
(H) $[\![\Delta, x]\!] \subseteq [\![\Gamma, x]\!]$, *for all* $x \in dom(\Gamma)$ ***(R)*** $[\![\Delta, v]\!] \subseteq [\![\Delta, e]\!]$

Using this theorem and with the aid of some simple auxiliary results Th. 3.2.3 can be easily proven. Now we can use it to obtain a completeness result for the original *CRWL* wrt. *FLC*.

Corollary 3.2.1 ([LRS07a] Corollary 6.6)**.** *Let* $\mathcal{P}$ *be a* $CRWL$*-program,* $\hat{\mathcal{P}}$ *its FLC-transformation,* e *a* $CRWL$*-expression, and* e^* *its normalization. Then* $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow \Delta : v$ *implies* $\lvert ligs(\Delta, v)\rvert \in [\![e]\!]^{\mathcal{P}}_{CRWL}$.

Note that whenever $\Delta : v$ corresponds to a normal form, i.e., the implicit graph represented in $\Delta : v$ corresponds to a c-term t, then $\lvert ligs(\Delta, v)\rvert = t$ and so $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ by this corollary.

Completeness of *FLC* wrt. *CRWL*
When we approach the completeness of *FLC* with respect to $CRWL_{FLC}$ we have to face again the problem that *FLC* stops evaluation at head normal forms. To overcome the problem we add to the set of rules defining the *FLC*-reduction relation $\Downarrow$ a new rule to continue evaluation inside heaps, namely the rule (Contx) in Fig. 3.8. We write $\Downarrow^{Ctx}$ for this new relation – clearly an extension of $\Downarrow$ – that goes beyond head normal forms.

$$\textbf{(Contx)} \quad \frac{\Gamma : x_i \Downarrow \Delta : v_i \qquad \Delta : e \Downarrow \Theta : v}{\Gamma : e \Downarrow \Theta : v} \quad \text{where } x_i \in dom(\Gamma)$$

Figure 3.8: The rule Contx for *FLC* [LRS07a]

The relation $\Downarrow^{Ctx}$ still satisfies Th. 3.2.4. This, together with the fact that $\exists\Delta$ such that $\Gamma : e \Downarrow \Delta : c(\overline{x})$ iff $\exists\Delta'$ such that $\Gamma : e \Downarrow^{Ctx} \Delta' : c(\overline{x})$, are enough to justify its use from now on. We have also found convenient to define a variation of $CRWL_{FLC}$ whose proofs are more similar to those for *FLC*, and call it $NCRWL_{FLC}$. This calculus is defined by replacing the rules **(DF)** and **(CASE)** in Figure 3.5 by those in Fig. 3.9. It can be

$$\frac{let\ \overline{x} = \overline{t}\ in\ e[\overline{y/t}] \to t}{f(\overline{t}) \to t}\ \textbf{(DFN)} \qquad \text{if } (f(\overline{y}) = e) \in \mathcal{P}, \text{ with } \overline{x} \text{ fresh}$$

$$\frac{e \to c(\overline{t}) \quad let\ \overline{y} = \overline{t}\ in\ e_i[\overline{x/y}] \to t}{case\ e\ of\ \{\overline{p_k \to e_k}\} \to t}\ \textbf{(CASEN)} \qquad \text{if } p_i = c(\overline{x}), \text{ with } \overline{y} \text{ fresh}$$

Figure 3.9: The new rules for $NCRWL_{FLC}$ [LRS07a]

easily proven that $CRWL_{FLC}$ and $NCRWL_{FLC}$ define the same denotations, simply by induction on the size of the proofs.

We will also use the notion of hyponormalized FLC expression, which assures that every argument of a constructor or function symbol present in an hyponormalized expression is a c-term. Now we can present our main result concerning the completeness of FLC with respect to $CRWL_{FLC}$:

Theorem 3.2.5 ([LRS07a] Th. 6.11)**.** *If $e \in Exp_\perp$ is hyponormalized and $t \in CTerm_\perp$, then:*

a) $\mathcal{P} \vdash_{CRWL_{FLC}} e \to c(\overline{t})$ *implies* $[] : e^* \Downarrow \Delta : c(\overline{x})$*, for some* $\Delta, \overline{x}$*.*

b) $\mathcal{P} \vdash_{CRWL_{FLC}} e \to t, t \neq \perp$ *implies* $[] : e^* \Downarrow^{Ctx} \Delta : t'$ *for some* Δ, t' *such that* $|ligs(\Delta, t')| \sqsupseteq t$

The first part *a)* states that FLC is able to obtain the outer constructor of the result of a $CRWL_{FLC}$-derivation. Part b), which is stronger, says that not only the outer constructor, but the whole result of a $CRWL_{FLC}$-derivation is covered by a FLC, if the information implicit in the heap is taken into account by means of the function $ligs$.

To prove this result we first obtain a similar one for the auxiliary calculus $NCRWL_{FLC}$:

Theorem 3.2.6 ([LRS07a] Th. 6.12)**.** *If $e \in Exp_\perp$ is hyponormalized and $\mathcal{P} \vdash_{NCRWL_{FLC}} e \to t$ with $t \neq \perp$, then $[] : e^* \Downarrow^{Ctx} \Delta : t'$ for some Δ, t' such that $|ligs(\Delta, t')| \sqsupseteq t$*

With the aid of this theorem the proof of Th. 3.2.5 comes pretty easily. The proof of Th. 3.2.6 is very involved and we refer the reader to [LRS07a] (Sect. 7.2.2, page 140) for details. Then, with Th. 3.2.5 at hand, we can prove the following completeness result of FLC wrt. the original $CRWL$ at last.

Corollary 3.2.2 ([LRS07a] Corollary 6.15)**.** *Let $\mathcal{P}$ be a $CRWL$-program, $\hat{\mathcal{P}}$ its FLC-transformation, e a $CRWL$-expression, and e^* its normalization. Then $\mathcal{P} \vdash_{CRWL} e \to t, t \neq \perp$ implies $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow^{Ctx} \Delta : t'$ such that $|ligs(\Delta, t')| \sqsupseteq t$.*

CRWL vs. FLC: conclusions

We have obtained an equivalence result for ground expressions and for the class of FLC-programs not having recursive *let* bindings nor extra variables. We think that this restricted case is interesting in itself, as a non-trivial technical basis for future generalizations. Furthermore along the way some interesting new notions for FLC like the novel $CRWL_{FLC}$ calculus have been developed, which could be useful for future works. Anyway the importance of those restrictions is somehow alleviated by the fact that extra variables can be safely removed from programs [dDL07, AH06], and recursive *let*'s do not appear in the

translation of *CRWL*-programs to *FLC*-syntax. Still, dropping the imposed restrictions is of course desirable, and we hope to do it in the next future.

The great difficulties encountered during the proofs, even with the imposed restrictions, suggest to look for new insights, not only at the level of the proofs but also in the sense of finding new alternative semantical descriptions of functional logic programs. We will deal with that subject in the next section.

3.2.2 *Let*-rewriting

In previous sections we have seen several formalisms that give a semantic description for FLP as it is implemented in modern systems like the *Toy* language [LS99, CSe06] or the variants of *Curry* [Han06]. Hence these formalisms describe a semantics for non strict functions with call-time choice, using CS's with extra variables as programs.
Although each of these descriptions can be the most suitable option for the analysis of programs in some situations, none of them provides a simple, precise and high level notion of reduction step in the evaluation of expressions: the rewriting logic *CRWL* works with denotations of expressions, and so it is far from the operational level, and although its associated goal-solving calculi [GHLR99, Vad03] provide a notion of step, they handle complex configurations so the simplicity of expressions used in the programs is lost; the *FLC* framework is based on program transformations used to encode a concrete evaluation strategy, and works on a very low abstraction level; the formalization of graph rewriting of [EJ97, EJ98] entails the high complexity of the manipulation of graph structures, which maybe could be avoided in a formalism specialized in the restricted class of TRS's used in FLP.

On the other hand term rewriting, which otherwise would be the obvious choice for this task, cannot be used to describe the semantics of FLP systems because it performs run-time choice parameter passing. But, would it be possible defining a program transformation that, accepting a *CRWL*-program as input, yields a TRS that evaluated under traditional term rewriting would behave as the input program under call-time choice? That is, there is a program transformation to perfectly mimic call-time choice within ordinary rewriting? The following (counter)example from [LRS07b] (Sect. 7.1.1, page 126) shows it is not the case, exploiting the fact that rewriting is closed under general substitutions while *CRWL*-provability is only closed under c-substitutions.

Example 3.2.1 ([LRS07b] Ex. 1)**.** Consider the program $\mathcal{P}$, expected to be interpreted under call-time choice:

$$f(X) \to c(X,X) \qquad coin \to 0 \qquad coin \to 1$$

and assume a TRS $\mathcal{P}'$ such that: $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \Leftrightarrow e \to^*_{\mathcal{P}'} t$, for all e, t. We will arrive to a contradiction. Since $\mathcal{P} \vdash_{CRWL} f(X) \twoheadrightarrow c(X,X)$, we must have $f(X) \to^*_{\mathcal{P}'} c(X,X)$. Now, since $\to^*_{\mathcal{P}'}$ is closed under substitutions, we have $f(coin) \to^*_{\mathcal{P}'} c(coin, coin)$, and then we have the reductions $f(coin) \to^*_{\mathcal{P}'} c(coin, coin) \to^*_{\mathcal{P}'} c(0,1)$. But it is easy to see that $\mathcal{P} \vdash_{CRWL} f(coin) \twoheadrightarrow c(0,1)$ does not hold.

This result was later developed in Th. 7 from [LRS09a] (Sect. 7.1.4, page 139), where it was shown that the converse perfect imitation, that is, expressing run-time choice within call-time choice, is not possible either, in this case due to different compositionality properties of both kind of choices. That result can be summarized in the following easy consequence of it.

Proposition 3.2.1. It is not true that for any program $\mathcal{P}$, expected to be interpreted under run-time choice, exists some program $\mathcal{P}'$ such that

$$\mathcal{P} \vdash e \to^* t \text{ iff } \mathcal{P}' \vdash_{CRWL} e \twoheadrightarrow t$$

for any $e \in Exp, t \in CTerm$

Another possibility could be imposing the restriction to term rewriting, allowing only substitutions to be used in rewriting steps, in a similar way as it happens in the OR rule of *CRWL*. But this would describe a strict semantics, as for example under the program $\mathcal{P} = \{f(X) \to 0,\ loop \to loop\}$ the expression $f(loop)$ could not be reduced to the value 0. Therefore we need another rule to reduce unnecessary subexpressions to a special c-term expressing absence of information, that is, to $\bot$. Since not-neededness is undecidable, this special reduction must be allowed for any subexpression. The result of this discussion is the one-step reduction relation given in Figure 3.10, first appearing in [LRS07b].

B^{rw}	$\mathcal{C}[e] \rightarrowtail \mathcal{C}[\bot]$	$\mathcal{C} \in Cntxt, e \in Exp_\bot$
OR^{rw}	$\mathcal{C}[f(t_1, \ldots, t_n)\theta] \rightarrowtail \mathcal{C}[r\theta]$	$\mathcal{C} \in Cntxt$, $(f(t_1, \ldots, t_n) \to r) \in \mathcal{P}$, $\theta \in CSubst_\bot$

Figure 3.10: *CRWL*-rewriting [LRS07b]

CRWL-rewriting is essentially equivalent to *CRWL*, as the following results shows.

Theorem 3.2.7 ([LRS07b] Th. 1)**.** *Let $\mathcal{P}$ be a CRWL-program, $e \in Exp_\bot, t \in CTerm_\bot$. Then $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ iff $e \rightarrowtail^*_{\mathcal{P}} t$.*

We remark that OR^{rw} essentially corresponds to innermost evaluation [BN98]. So the result has the following interesting reading: non-strict call-time choice can be achieved via innermost evaluation if at any step one has the possibility of reducing a subexpression to $\bot$. In this line we could describe the kind of parameter passing performed by OR^{rw} as "call-by-partial-value".

Example 3.2.2. Consider the program of Ex. 3.1.1 (page 21) again, a $\rightarrowtail$-rewrite sequence for the expression $heads(repeat(coin))$ could be:

$$heads(repeat(coin)) \rightarrowtail heads(repeat(0)) \rightarrowtail heads(0 : repeat(0))$$
$$\rightarrowtail heads(0 : 0 : repeat(0)) \rightarrowtail heads(0 : 0 : \bot) \rightarrowtail (0, 0)$$

The rules for $\rightarrowtail$ can actually serve for a very easy implementation of non-strict call-time choice, but with a major drawback: reduction follows an unnatural order and requires, at any step, an unavoidable guessing between the two rules B^{rw} and OR^{rw}, leading to high inefficiency. Therefore, $\rightarrowtail$ achieves only partially our goals and we cannot consider it as the natural reduction notion we are looking for.

Nevertheless, this difficulties were overcome through the notion of *let*-rewriting, which we will introduce here as it was first presented in [LRS07b] (Sect. 7.1.1, page 126). In this modification of term rewriting inspired in [AFM+95, MOW98, Plu99, SH04], the notions of subexpression sharing and call-time choice are added to the framework of term rewriting. In later sections we will also present the corresponding narrowing relation, and extensions to higher order settings.

Syntax of *let*-rewriting

The basic idea of *let*-rewriting is extending the syntax of expressions by adding local bindings through a new *let* construct, to be able to reason about sharing of subexpressions. Formally the syntax for *let-expressions* is:

$$LExp \ni e ::= X \mid h(\overline{e}) \mid let\ X = e_1\ in\ e_2$$

where $X \in \mathcal{V}$, $h \in CS \cup FS$, $\overline{e}$ is a tuple of *let*-expressions, and e_1, e_2 are single *let*-expressions. We will use the notation $let\ \overline{X = a}\ in\ e$ as a shortcut for $let\ X_1 = a_1\ in \ldots in\ let\ X_n = a_n\ in\ e$. The notion of *one-hole context* is also extended to the new syntax:

$$\mathcal{C} ::= [\,] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots, \mathcal{C}, \ldots)$$

The sets $FV(e)$ of *free* and $BV(e)$ *bound* variables of $e \in LExp$ are defined as $FV(X) = \{X\}$; $FV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} FV(e_i)$; $FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\})$; $BV(X) = \emptyset$; $BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i)$; $BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\}$. Notice that with the given definition of $FV(let\ X = e_1\ in\ e_2)$ there are not recursive *let*-bindings in the language since the possible occurrences of X in e_1 are not considered bound and therefore refer to a 'different' X. This is similar to what is done in [MOW98, AFM+95], but not in [AHH+05, Lau93].

The *let*-rewriting relation that we will present in the next section is devised to handle *CRWL*-programs as those introduced in Sect. 3.1.2 (page 22), and in particular extra variables are allowed in program rules.

Rules of *let*-rewriting

The *let*-rewriting relation $\rightarrow_l$ is shown in Figure 3.21. Rule Contx allows us to use any subexpression as redex in the derivation. Fapp performs a rewriting step in the proper sense, using a program rule. Note that only c-substitutions are allowed, to avoid copying of unevaluated expressions which would destroy sharing and call-time choice. To avoid a strict semantics we also have the rule LetIn, used to suspend the evaluation of a subexpression by introducing a *let* binding. The advantage of LetIn when compared to B^{rw} of Fig. 3.10 (page 39) is that we do not lose information in the suspension. If the suspended expression is later needed its evaluation can be performed by some Contx steps and then its result propagated by Bind. This later rule is safe wrt. call-time choice because it only propagates c-terms, that is, either completely defined values (without any bound variable) or partially computed values with some suspension (bound variable) on it, which will be safely managed by the calculus. On the other hand, if the bound variable disappears from the body of the *let* during evaluation, rule Elim can be used for garbage collection. This rule is needed because we want normal forms corresponding to values to be c-terms. Finally rule Flat is needed for flattening nested *let*s, otherwise some reductions could become wrongly blocked or forced to diverge. Note that *let*-rewriting does not need to use the semantic value $\bot$, being this one of its fundamental differences with *CRWL*, which relies on $\bot$ for getting a non-strict semantics.

Example 3.2.3. As a complete derivation example, consider the program of Ex. 3.1.1 (page 21) again and the following reduction from $heads(repeat(coin))$ to $(0, 0)$. Notice that

Contx	$\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $e \to_l e'$, $\mathcal{C} \in Cntxt$
LetIn	$h(\ldots, e, \ldots) \to_l let\ X = e\ in\ h(\ldots, X, \ldots)$ if $h \in CS \cup FS$, e takes one of the forms $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable
Flat	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ assuming that Y does not appear free in e_3
Bind	$let\ X = t\ in\ e \to_l e[X/t]$, if $t \in CTerm$
Elim	$let\ X = e_1\ in\ e_2 \to_l e_2$, if X does not appear free in e_2
Fapp	$f(t_1\theta, \ldots, t_n\theta) \to_l e\theta$, if $f(t_1, \ldots, t_n) \to e \in \mathcal{P}$, $\theta \in CSubst$

Figure 3.11: Rules of *let*-rewriting [LRS07b]

there is not a unique $\to_l$-reduction leading to $(0, 0)$. The definition of $\to_l$ does not prescribe any particular strategy, as it has been designed to be compatible with any rewriting strategy, as it happens with term rewriting.

$heads(repeat(coin)) \to_l let\ X = repeat(coin)\ in\ heads(X)$ LetIn
$\to_l let\ X = (let\ Y = coin\ in\ repeat(Y))\ in\ heads(X)$ LetIn
$\to_l let\ Y = coin\ in\ let\ X = repeat(Y)\ in\ heads(X)$ Flat
$\to_l let\ Y = coin\ in\ let\ X = Y : repeat(Y)\ in\ heads(X)$ Fapp
$\to_l let\ Y = coin\ in\ let\ X = (let\ Z = repeat(Y)\ in\ Y : Z)\ in\ heads(X)$ LetIn
$\to_l let\ Y = coin\ in\ let\ Z = repeat(Y)\ in\ let\ X = Y : Z\ in\ heads(X)$ Flat
$\to_l let\ Y = coin\ in\ let\ Z = repeat(Y)\ in\ heads(Y : Z)$ Bind
$\to_l let\ Y = coin\ in\ let\ Z = Y : repeat(Y)\ in\ heads(Y : Z)$ Fapp
$\to_l let\ Y = coin\ in\ let\ Z = (let\ U = repeat(Y)\ in\ Y : U)\ in\ heads(Y : Z)$ LetIn
$\to_l let\ Y = coin\ in\ let\ U = repeat(Y)\ in\ let\ Z = Y : U\ in\ heads(Y : Z)$ Flat
$\to_l let\ Y = coin\ in\ let\ U = repeat(Y)\ in\ heads(Y : Y : U)$ Bind
$\to_l let\ Y = coin\ in\ let\ U = repeat(Y)\ in\ (Y, Y)$ Fapp
$\to_l let\ Y = coin\ in\ (Y, Y)$ Elim
$\to_l let\ Y = 0\ in\ (Y, Y)$ Fapp
$\to_l (0, 0)$ Bind

Adequacy of *let*-rewriting

In this section we will review the soundness and completeness results of *let*-rewriting with respect to *CRWL* presented in [LRS07b] (Sect. 7.1.1, page 126). To this purpose we will need to consider $\bot$ at some points. Therefore we define the set $LExp_\bot$ in the natural way. We also extend the notion of shell of Sect. 3.1.2 to $LExp_\bot$ as $|let\ X = e_1\ in\ e_2| = |e_2|[X/|e_1|]$. Notice that the information contained in *let*-bindings is taken into account for building up the shell of *let*-expressions.

Soundness Concerning soundness we would like to prove something like this:

$$If\ e \to_l e'\ then\ [\![e']\!]_{CRWL} \subseteq [\![e]\!]_{CRWL}, \text{ for any } e, e' \in Exp.$$

That is, $\to_l$-steps do not create new *CRWL*-semantic values. But *let*-expressions are not defined in *CRWL* and even if we start with an expression without *lets*, *let*-rewriting may

introduce them by LetIn. To cope with this situation we enlarge the *CRWL*-calculus in Figure 3.1 (page 22) to a new calculus $CRWL_{let}$, by adding a new rule for dealing with *let*-expressions, as it was done in $CRWL_{FLC}$—see Fig. 3.5 (page 34).

$$\textbf{Let}\ \frac{e_1 \twoheadrightarrow t_1 \quad e[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e \twoheadrightarrow t}$$

With the aid of $CRWL_{let}$, our first soundness result is stated as follows.

Theorem 3.2.8 (One-Step Soundness of *let*-rewriting [LRS07b] Th. 2)**.**
For any $e, e' \in LExp$,

$$e \to_l e' \ \textit{implies}\ [\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}.$$

Notice that because of non-determinism $\subseteq$ cannot be replaced by $=$ in this Theorem. The proof of Th. 3.2.8 is complicated by the fact that the following *monotonocity under contexts* property of $CRWL_{let}$ does not hold for any context $\mathcal{C}$:

$$[\![e]\!]_{CRWL_{let}} \subseteq [\![e']\!]_{CRWL_{let}} \text{ implies } [\![\mathcal{C}[e]]\!]_{CRWL_{let}} \subseteq [\![\mathcal{C}[e']]\!]_{CRWL_{let}}$$

Unfortunately this property is false because of the possible capture of variables when switching from e to $\mathcal{C}[e]$, as the following example shows:

Example 3.2.4. If f is defined by just the rule $f(0) \to 1$ then we have

$$\{\bot\} \equiv [\![f(X)]\!] \subseteq [\![0]\!] \equiv \{\bot, 0\}$$

but when these expressions are placed within the context $let\ X = 0\ in\ [\ ]$ we obtain

$$\{\bot, 1\} \equiv [\![let\ X = 0\ in\ f(X)]\!] \not\subseteq [\![let\ X = 0\ in\ 0]\!] \equiv \{\bot, 0\}.$$

To overcome this problem we define the stronger notion of *hypersemantics*, which gives a more active role to variables in the expression; it is the function $[\![e]\!]^{\mathcal{P}} : CSubst_{\bot} \to \mathcal{P}(CTerm_{\bot})$ defined by $[\![e]\!]^{\mathcal{P}}\theta = [\![e\theta]\!]^{\mathcal{P}}$. Hypersemantics are useful to characterize the meaning of expressions present in a context in which some of its variables may get bound, like in the body of a *let* or in the right hand side of a program rule. As a result, hypersemantics fulfils the desired monotonicity property, using the hypersemantics order defined as $[\![e]\!] \Subset [\![e']\!]$ iff $\forall\theta.[\![e]\!]\theta \subseteq [\![e']\!]\theta$.

Lemma 3.2.1 (Monotonicity of hypersemantics under contexts, [LRS07b] Lemma 4)**.** *For any* $e, e' \in LExp_{\bot}$, *and every context* $\mathcal{C}$ *we have:*

$$[\![e]\!]_{CRWL_{let}} \Subset [\![e']\!]_{CRWL_{let}} \ \textit{implies}\ [\![\mathcal{C}[e]]\!]_{CRWL_{let}} \Subset [\![\mathcal{C}[e']]\!]_{CRWL_{let}}$$

Using this and some additional auxiliary results we can prove a generalization of Th. 3.2.8 to hypersemantics, which becomes a trivial corollary of it.

Theorem 3.2.9 (One-Step Hyper-Soundness of *let*-rewriting, [LRS07b] Th. 3)**.**
For any $e, e' \in LExp$

$$e \to_l e' \ \textit{implies}\ [\![e']\!]_{CRWL_{let}} \Subset [\![e]\!]_{CRWL_{let}}$$

And with a little additional development we arrive to our main result concerning the soundness of *let*-rewriting wrt. *CRWL*.

Theorem 3.2.10 (Soundness of *let*-rewriting, [LRS07b] Th. 4)**.**
Let $\mathcal{P}$ *be a CRWL-program and* $e \in Exp$. *Then:*
(i) $e \to_l^* e'$ *implies* $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow |e'|$, *for any* $e' \in LExp$.
(ii) $e \to_l^* t$ *implies* $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, *for any* $t \in CTerm$.

Completeness Now we look for the reverse implication of Theorem 3.2.10. First of all we need the following auxiliary lemma.

Lemma 3.2.2 (Completeness lemma for *let*-rewriting, [LRS07b] Lemma 8)**.** *Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm_\bot$ such that $t \neq \bot$. Then:*

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \text{ implies } e \rightarrow_l^* \text{let } \overline{X = a} \text{ in } t'$$

for some $t' \in CTerm$ and $\overline{a} \subseteq LExp$ in such a way that $t \sqsubseteq |\text{let } \overline{X = a} \text{ in } t'|$ and $|a_i| = \bot$ for all $a_i \in \overline{a}$. As a consequence, $t \sqsubseteq t'[\overline{X/\ \bot}]$.

Using this lemma and a small additional effort we get the following strong completeness result.

Theorem 3.2.11 (Completeness of *let*-rewriting, [LRS07b] Th. 5)**.**
Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm_\bot$. Then:

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \text{ implies } e \rightarrow_l^* e'$$

for some $e' \in LExp$ such that $t \sqsubseteq |e'|$.

And now we can combine our soundness and completeness results to obtain a strong equivalence result for both formalisms:

Theorem 3.2.12 (Equivalence of *CRWL* and *let*-rewriting, [LRS07b] Th. 7)**.**
Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm$. Then:

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \text{ iff } e \rightarrow_l^* t.$$

***Let*-rewriting: conclusions**

With the definition of *let*-rewriting we have tried to provide a simple, precise and high level notion of reduction step in the evaluation of expressions in FLP under a call-time choice semantics. Our aim has been to fill a gap existing in the FLP field, which is the technical disconnection between two of the most accepted approaches to the paradigm: one, given by the *CRWL* framework, more biased to the semantics; and the other, focused in operational aspects like the definition of on-demand evaluation strategies, based on the theory or term rewriting.

Finally the obtained *let*-rewriting relation is nothing more than a simple textual formulation of term graph rewriting. This simplicity has been achieved by taking advantage of the restrictions imposed in the kind of programs managed by *let*-rewriting, that is, left linear constructor based TRS's with extra variables, and of the absence of cyclic bindings. But this kind of programs are precisely those used in modern FLP systems, so as a consequence the resulting framework is very convenient for reasoning about FLP computations from a high abstraction level, without having to deal with a highly complex manipulation of graph structures. Anyway we refer the interested reader to Sect. 7 of [LRS07b] for a discussion about related work.

We must warn that *let*-rewriting as presented here does not pretend to be in its own the working operational procedure for FLP programs, for several reasons: first, we have not considered any on-demand evaluation strategy, which is something needed in practice, otherwise the seach space turns out to be too large. Second, there are two situations

in computations where rewriting is not enough and must be lifted to narrowing: when the program uses extra variables (narrowing must be used then to obtain their values; rewriting 'magically' guesses them in the parameter passing substitution) and when the initial expression to reduce has variables. The extension of this work to deal with narrowing, as well as the addition of higher order features, will be presented in subsequent sections.

3.2.3 *Let*-narrowing

Taking into account that the implementation of modern FLP systems like *Toy* or *Curry* are based on narrowing as its basic operational procedure (combined with residuation in the case of *Curry*), a natural extension of the framework of *let*-rewriting has been defining the corresponding *let*-narrowing relation [LRS09d] (Sect. 7.2.1, page 140). By doing this we have finally obtained a very simple and high level stepwise operational description of functional-logic computations.

We still have kept *let*-narrowing independent of any particular on-demand strategy like for example needed narrowing [AEH94] or natural narrowing [EMT05], and by doing this we have also made it compatible with any of these strategies, which could be attached to *let*-narrowing thus defining several on-demand versions of *let*-narrowing, an interesting possible subject of future work. The fact that current major implementations of FLP are based on needed narrowing, whose optimality and general theory has been formulated for term rewriting, which is unsound for call-time choice, motivates the interest of *let*-narrowing and its possible on-demand extensions.

The standard definition of *narrowing* as a lifting of rewriting in ordinary TRS's says (adapted to the notation of contexts): $\mathcal{C}[f(\overline{t})] \leadsto_\theta \mathcal{C}\theta[r\theta]$, if θ is a mgu of $f(\overline{t})$ and $f(\overline{s})$, where $f(\overline{s}) \to r$ is a fresh variant of a rule of the TRS. We note that frequently the narrowing step is not decorated with the whole unifier θ, but with its projection over the variables in the narrowed expression.

This definition of narrowing cannot be directly translated as it is to the case of *let*-rewriting, for two important reasons. The first is not new: because of call-time choice, binding substitutions must be c-substitutions, as already happened in *let*-rewriting. The second is that produced variables (those introduced by LetIn and bound in a *let* construction) should not be narrowed, because their role is to express intermediate values that are evaluated at most once and shared, according to call-time choice. Therefore the value of produced variables should be better obtained by evaluation of their binding expressions, and not by bindings coming from narrowing steps. Furthermore, to narrow on produced variables destroys the structure of *let*-expressions. The following example illustrates some of the points above.

Example 3.2.5 ([LRS09d] Ex. 1)**.** Consider the following program over natural numbers (represented with constructors 0 and s):

$$
\begin{array}{ll}
0 + Y \to Y & even(X) \to if\ (Y + Y == X)\ then\ true \\
s(X) + Y \to s(X + Y) & if\ true\ then\ Y \to Y \\
0 == 0 \to true & s(X) == s(Y) \to X == Y \\
0 == s(Y) \to false & s(X) == 0 \to false \\
coin \to 0 & coin \to s(0)
\end{array}
$$

Notice that the rule for *even* has an extra variable Y. With this program, the evaluation

of *even(coin)* by *let*-rewriting could start as follows:

$$\begin{array}{l} even(coin) \to_l let\ X = coin\ in\ even(X) \\ \to_l let\ X = coin\ in\ if\ (Y + Y == X)\ then\ true \\ \to_l^* let\ X = coin\ in\ let\ U = Y + Y\ in\ let\ V = (U == X)\ in\ if\ V\ then\ true \\ \to_l^* let\ U = Y + Y\ in\ let\ V = (U == 0)\ in\ if\ V\ then\ true \end{array}$$

Now, all function applications involve variables and therefore narrowing is required to continue the evaluation. But notice that if we perform classical narrowing in (for instance) *if V then true*, then the binding $\{V/true\}$ is created. What to do with this binding in the surrounding context? If the binding is textually propagated to the whole expression, we obtain

$$let\ U{=}Y{+}Y\ in\ let\ true{=}(U{==}0)\ in\ true$$

which is not a legal expression of $LExp$ (due to the binding *let true =(U==0)*). If, as it could seem more correct by our variable convention, we rename the V occurring as produced in the the context, we would obtain

$$let\ U{=}Y{+}Y\ in\ let\ W{=}(U{==}0)\ in\ true$$

losing the connection between V and *true*; as a result, the expression above reduces now to *true* only 'by chance', and the same result would have been obtained also if *coin* had been reduced to $s(0)$ instead of 0, which is obviously wrong.

A similar situation would arise if narrowing was done in $U == 0$, giving the binding $\{U/0\}$. The problem in both cases stems from the fact that V, U are bound variables. What is harmless is to perform narrowing in $Y + Y$ (Y is a free variable). This gives the binding $\{Y/0\}$ and the result 0 for the subexpression $Y + Y$. Put in its surrounding context, the derivation above would continue as follows:

$$\begin{array}{l} let\ U = 0\ in\ let\ V = (U == 0)\ in\ if\ V\ then\ true \\ \to_l let\ V = (0 == 0)\ in\ if\ V\ then\ true \\ \to_l let\ V = true\ in\ if\ V\ then\ true \\ \to_l if\ true\ then\ true \to_l true \end{array}$$

The previous example shows that *let*-narrowing *must protect produced variables* against bindings. To express this we could add to the narrowing relation a parameter containing the set of protected variables. Instead of that, we have found more convenient to consider a distinguished set $PVar \subset \mathcal{V}$ of *produced variables* notated as $X_p, Y_p, \ldots$, to be used according to the following criteria: variables bound in a *let* expression must belong to $PVar$ (therefore *let* expressions have the form *let* X_p*=e in e'*); program rules (and fresh variants of them) do not use variables of $PVar$; the parameter passing c-substitution θ in the rule **FAPP** of *let*-rewriting replaces extra variables in the rule by c-terms not having variables of $PVar$; and rewriting (or narrowing) sequences start with initial expressions e not having free occurrences of produced variables (i.e., $FV(e) \cap PVar = \emptyset$). Furthermore we will need the following notion:

Definition 3.2.2 (Admissible substitutions)**.** A substitution θ is called *admissible* iff $\theta \in CSubst$ and $(dom(\theta) \cup vran(\theta)) \cap PVar = \emptyset$.

Admissible substitutions do not interfere with produced variables, and play a role in the results relating *let*-narrowing and *let*-rewriting that can be found below.

Rules of *let*-narrowing

The one-step *let*-narrowing relation $e \leadsto^l_\theta e'$ (assuming a given program $\mathcal{P}$) is defined in Figure 3.12. The rules ELIM, BIND, FLAT, LETIN of *let*-rewriting are kept untouched except for the decoration with the empty substitution ϵ.

CONTX $\mathcal{C}[e] \leadsto^l_\theta \mathcal{C}\theta[e']$ if $e \leadsto^l_\theta e'$, $\mathcal{C} \in Cntxt$

NARR $f(\overline{t}) \leadsto^l_{\theta|_{FV(f(\overline{t}))}} r\theta$, for any fresh variant $(f(\overline{p}) \to r) \in \mathcal{P}$ and $\theta \in CSubst$ such that:

i) $f(\overline{t})\theta \equiv f(\overline{p})\theta$.

ii) $dom(\theta) \cap PVar = \emptyset$.

iii) $vran(\theta|_{\setminus FV(f(\overline{p}))}) \cap PVar = \emptyset$.

X $e \leadsto^l_\epsilon e'$ if $e \to_l e'$ using $\boldsymbol{X} \in$ {ELIM, BIND, FLAT, LETIN}.

Figure 3.12: Rules of *let*-narrowing [LRS09d]

The rule **NARR** requires some explanations:

- We impose $\theta \in CSubst$ to ensure that call-time choice is respected, as in the rule FAPP of *let*-rewriting.
- The condition *(i)* simply expresses that θ is a unifier of $f(\overline{t})$ and $f(\overline{p})$. To avoid unnecessary loss of generality or applicability of our approach, we do not impose θ to be a mgu.
- Condition *(iii)* is a subtle one stating that the bindings in θ for extra variables and for variables in the expression being narrowed do not introduce produced variables. Otherwise, undesired situations arise, when NARR is combined with CONTX. Consider for instance, the program rules $f \to Y$ and $loop \to loop$ and the expression $let\ X_p = loop\ in\ f$. This expression could be reduced in the following way:

$$let\ X_p = loop\ in\ f \leadsto^l_\epsilon let\ X_p = loop\ in\ Z$$

by applying NARR to f with $\theta = \epsilon$ taking the fresh variant rule $f \to Z$, and using CONTX for the whole expression. If we drop condition *(iii)* we could perform a similar derivation using the same fresh variant of the rule for f, but now using the substitution $\theta = \{Z/X_p\}$:

$$let\ X_p = loop\ in\ f \leadsto^l_\epsilon let\ X_p = loop\ in\ X_p$$

which is certainly not intended because the free variable Z in the previous derivation appears now as a produced variable, i.e., we get an undesired capture of variables.

On the other hand, we also remark that if the substitution θ in NARR is chosen to be a standard mgu[3] of $f(\overline{t})$ and $f(\overline{p})$ (which is always possible) then the condition *(iii)* is always fulfilled.

[3]By standard mgu of t, s we mean an idempotent mgu θ with $dom(\theta) \cup vran(\theta) \subseteq var(t) \cup var(s)$.

- Notice finally that in a NARR-step not the whole θ is recorded, but only its projection over the relevant variables, i.e. the variables in the narrowed expression. This, together with *(i)* and *(ii)*, guarantees that the annotated substitution is an admissible one, a property not fulfilled in general by the whole θ, since the parameter passing (one of the roles of θ) might bind variables coming from the program rule to terms containing produced variables (as for example with the program $\{f(X) \to 0, loop \to loop\}$ and the step $let\ X_p = loop\ in\ f(X_p) \leadsto^l_\epsilon let\ X_p = loop\ in\ 0$, which uses $\theta = \{X/X_p\}$).

The one-step relation $\leadsto^l_\theta$ is extended in the natural way to the multiple-steps narrowing relation $\leadsto^{l^*}$, which is defined as the least reflexive relation verifying:

$$e \leadsto^l_\epsilon e \qquad e \leadsto^l_{\theta_1} e_1 \leadsto^l_{\theta_2} \ldots e_n \leadsto^l_{\theta_n} e' \ \Rightarrow\ e \leadsto^{l^*}_{\theta_1 \ldots \theta_n} e'$$

We write $e \leadsto^{l^n}_\theta e'$ for a n-steps narrowing sequence.

Example 3.2.6 ([LRS09d] Ex. 2)**.** The following example shows how we can deal with the derivation of Ex. 3.2.5 using *let*-narrowing. In each NARR the narrowed expression is underlined.

$even(coin) \leadsto^l_\epsilon$	LETIN
$let\ X_p = coin\ in\ \underline{even(X_p)} \leadsto^l_\epsilon$	NARR
$let\ X_p = coin\ in\ if\ Y + Y == X_p\ then\ true \leadsto^{l^3}_\epsilon$	LETIN^2, FLAT
$let\ X_p = \underline{coin}\ in\ let\ U_p = Y + Y\ in$	
$\quad let\ V_p = (U_p == X_p)\ in\ if\ V_p\ then\ true \leadsto^l_\epsilon$	NARR
$let\ X_p = 0\ in\ let\ U_p = Y + Y\ in$	
$\quad let\ V_p = (U_p == X_p)\ in\ if\ V_p\ then\ true \leadsto^l_\epsilon$	BIND
$let\ U_p = \underline{Y+Y}\ in\ let\ V_p = (U_p == 0)\ in\ if\ V_p\ then\ true \leadsto^l_{\{Y/0\}}$	NARR
$let\ U_p = 0\ in\ let\ V_p = (U_p == 0)\ in\ if\ V_p\ then\ true \leadsto^l_\epsilon$	BIND
$let\ V_p = \underline{(0 == 0)}\ in\ if\ V_p\ then\ true \leadsto^l_\epsilon$	NARR
$let\ V_p = true\ in\ if\ V_p\ then\ true \leadsto^l_\epsilon$	BIND
$if\ true\ then\ true \leadsto^l_\epsilon$	NARR
$true$	

Adequacy of *Let*-narrowing

In this section we show the adequacy of *let*-narrowing wrt. *let*-rewriting.

Soundness Our main soundness result can be easily achieved by combining a closedness property of *let*-rewriting with some properties of admissible substitutions, and it is stated at follows.

Theorem 3.2.13 (Soundness of *let*-narrowing, [LRS09d] Th. 1)**.** *For any $e, e' \in LExp$, $e \leadsto^{l^*}_\theta e'$ implies $e\theta \to^*_l e'$.*

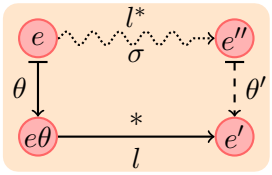
Note that by applying this result to Ex. 3.2.6 we conclude that there must exist a derivation $even(coin) \to^*_l true$, as $even(coin)$ is ground. This derivation could have the form:

$even(coin) \to_l$	LETIN
$let\ X_p = coin\ in\ even(X_p) \to_l$	FAPP
$let\ X_p = coin\ in\ if\ (0 + 0 == X_p)\ then\ true \to_l$	
$\ldots\ldots\ldots \to_l true$	

The indicated FAPP step in this *let*-rewriting derivation has used the substitution $\{Y/0\}$, thus anticipating and 'magically guessing' the right value of the extra variable Y of the rule of *even*. This kind of free instantiation of extra variables is also performed in the variation with extra variables of traditional term rewriting. In contrast, in the *let*-narrowing derivation the binding for Y is not done while reducing $even(X)$ but in a later NARR step over $Y + Y$, and the choice of the binding for Y is directed by the program rules for $+$. This is why *let*-narrowing is an effective operational procedure even in the presence of extra variables while it is not the case for *let*-rewriting, for which this 'magically guessing' is sometimes unavoidable. This way of organizing computations corresponds closely to the behavior of narrowing-based systems like *Toy* or *Curry*, and is the reason why *let*-narrowing is a good operational model for them.

Completeness Completeness is, as usual, more complicated to prove. The key result is the following generalization of Hullot's *lifting lemma* for classical rewriting and narrowing [Hul80], to the framework of *let*-rewriting. This lemma states that any rewrite sequence for a particular instance of an expression can be generalized by a narrowing derivation.

Lemma 3.2.3 (Lifting lemma for *let*-rewriting, [LRS09d] Lemma 2)**.** *Let $e, e' \in LExp$ such that $e\theta \to_l^* e'$ for an admissible θ, and let $\mathcal{W}$ be a finite set of variables with $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$. Then there exist a* let*-narrowing derivation $e \leadsto_\sigma^{l^*} e''$ and an admissible θ' such that $e''\theta' = e'$ and $\sigma\theta' = \theta[\mathcal{W}]$. Besides, the* let*-narrowing derivation can be chosen to use mgu's at each* NARR *step. Graphically:*



Now, combining this result with Th. 3.2.13 we obtain a strong adequacy theorem for *let*-narrowing with respect to *let*-rewriting.

Theorem 3.2.14 (Adequacy of *let*-narrowing, [LRS09d] Th. 2)**.**
Let $e, e_1 \in LExp$ and θ an admissible c-substitution, then:

$$e\theta \to_l^* e_1 \;\Leftrightarrow\; \begin{array}{l} \textit{there exists a let-narrowing derivation } e \leadsto_\sigma^{l^*} e_2 \\ \textit{and an admissible } \theta' \textit{ such that } \sigma\theta' = \theta[FV(e)],\ e_2\theta' \equiv e_1 \end{array}$$

Let-narrowing: conclusions

In this section we have continued our work about *let* rewriting by proposing the novel notion of *let*-narrowing, in which subexpression sharing is added to the traditional notion of narrowing for CS's. We have also proven that *let*-narrowing is sound and complete with respect to *let*-rewriting. We think that *let*-narrowing is the simplest proposed notion of narrowing that is close to the usual notions of TRS's and at the same time is adequate for call-time choice semantics. The main technical insight for *let*-narrowing has been the need of protecting produced (locally bound) variables against narrowing over them.

A natural extension of our work would be adding strategies to *let*-rewriting and *let*-narrowing, an issue still not tackled by us but which it is needed as a foundation for effective implementations of FLP. Nevertheless we think that the clear script we have followed so

far—first presenting a notion of rewriting with respect to which we have been able to prove correctness and completeness of a subsequent notion of narrowing, to which add strategies in future work—is an advantage rather than a lack of our approach. Keeping both *let*-rewriting and *let*-narrowing independent from a particular evaluation strategy puts these notions at a higher abstraction level which makes them better tools for reasoning about general properties of the language (see sections 3.2.4 and 3.2.5), or for comparisons with different semantics or semantic descriptions (as in sections 3.2.1 and 3.3.2). Then if we adapted an evaluation strategy to *let*-rewriting or *let*-narrowing, and proved its adequacy, we could also use the more general properties proved in the strategy-independent versions.

Another natural extension of this work would be the addition of higher order capabilities to our framework. We will deal with that matter in the next section.

3.2.4 *HOLet* Rewriting and Narrowing

One of the main features of functional logic languages is the use of higher order (HO) functions, a characteristic inherited from functional languages which maybe constitutes the most important sign of identity of that paradigm. However, most of the theoretical work about FLP focuses on first order (FO) aspects of programs, thus limiting the applicability of results. This is not a satisfactory situation, especially taking into account that the presence of functions that are at the same time HO and non-deterministic leads to somehow surprising behaviors, as shown by this example that we originally sent to the *Curry* mailing list [LF07]:

Example 3.2.7 ([LRS08a] Ex. 1)**.** Consider the following program computing with natural numbers represented by the constructors 0 and $s/1$, and where $+$ is defined as usual.

```
g X -> 0          f -> g          f' X -> f X
h X -> s 0        f -> h

fadd F G X -> (F X) + (G X)       fdouble F -> fadd F F
```

Notice that f and f' are non-deterministic functions that are (by definition of f') extensionally equivalent; from the point of view of standard functional programming they should be seen as 'the same function'. However, consider the expressions *fdouble f 0* and *fdouble f' 0*. In modern FLP languages like *Curry* [Han06] or *Toy* [LS99], the possible values for *fdouble f 0* are *0, s (s 0)*, while *fdouble f' 0* can be in addition reduced to *s 0*.

This behavior corresponds to the adaptation of call-time parameter passing to a higher order setting. The example was sent[4] to point out that η-expansion and η-reduction are not valid for such systems, because extensionally equivalent functions (e.g., f and f') can be semantically distinguishable when put in the same context (e.g., *double* [ ] 0), a fact that does not happen neither in standard (i.e, deterministic) functional programs[5], nor in FO FLP. We remark also that with *run-time choice* [Hus93, GHLR99], f and f' will be indistinguishable (*double f 0* and *double f' 0* would both produce *0, s 0, s (s 0)* as possible results). Therefore, it is the combination *HO + Non-determinism + call-time choice* which makes things different.

[4] As far as we know, it was the first time that this behavior was noticed.

[5] Although the addition of primitive functions not definable in the language like *seq* in Haskell [PJ03] can also destroy extensionality.

That combination was addressed in *HOCRWL* [GHR97, GHR01], an extension to higher order of *CRWL*. *HOCRWL* provides logic and model-theoretic semantics, based on an *intensional* view of functions, where different descriptions –in the form of *HO-patterns*– of the same extensional function are distinguished as different data. This allows expressive programs and is simpler than λ-calculus-based HO unification, which is an alternative approach followed in the logic programming setting [Mil91]. Previous work on the intensional view of HO-FLP [GHR92] did not consider non-determinism. Other works covering HO in FLP, [NMI95, HP99], consider orthogonal or inductively sequential (henceforth deterministic) systems; if extended directly to the non-deterministic case, they would realize run-time choice, as happens also with [AT99], where a type-based translation to FO in the spirit of [War82, Gon93] is proposed. We remark also that [HP99] is close to the theory of HO rewriting [TeR03], and therefore has η-expansion as a valid procedure, against the expected properties of the languages considered by ours. Finally, [AHH+05] copes with call-time choice but their approach to HO is again based on a FO-translation, in contrast to that of *HOCRWL*.

In this section we will present the extensions to higher order of *let*-rewriting and *let*-narrowing, as they appeared in [LRS08a] (Sect. 7.1.2, page 126) for the first time. As in the first order versions, the aim of this work is to provide a clear, simple notion of one-step reduction in the evaluation of FLP expressions under call-time choice parameter passing, in this case adapted to an higher order setting. As we saw above the combination of higher order, non-determinism and call-time choice gives rise to new challenges, and therefore these extensions to HO will not be trivial at all.

HOCRWL

Here we present some basic notions about *HOCRWL* [GHR97], as well as some new results [LRS08a].

Expressions, patterns and programs The set of *applicative expressions* is defined by $Exp \ni e ::= X \mid h \mid (e_1\ e_2)$. As usual, application is left associative and outer parentheses can be omitted, so that $e_1\ e_2\ \ldots e_n$ stands for $((\ldots(e_1\ e_2)\ldots)\ e_n)$. A distinguished set of expressions is that of *patterns* $t, s \in Pat$, defined by: $t ::= X \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_m$, where $0 \leq n \leq ar(c), 0 \leq m < ar(f)$. Patterns are irreducible expressions playing the role of *values*. *FO-patterns*, defined by $FOPat \ni t ::= X \mid c\ t_1 \ldots t_n\ (n = ar(c))$, correspond to FO constructor terms, representing ordinary non-functional data-values. Partial applications of symbols $h \in FS \cup CS$ to other patterns are HO-patterns and can be seen as truly data-values representing functions from an *intensional* point of view. Examples of patterns with the signature of Ex. 3.2.7 are: *0, s X, s, f', fadd f' f'*. The last three are HO-patterns. Notice that *f, fadd f f* are not patterns since f is not a pattern $(ar(f) = 0)$.

Expressions $X\ e_1 \ldots e_m$ $(m \geq 0)$ are called *flexible* (*variable application* when $m > 0$). *Rigid* expressions have the form $h\ e_1 \ldots e_m$; moreover, they are *junk* if $h \in CS^n$ and $m > n$, *active* if $h \in FS^n$ and $m \geq n$, and *passive* otherwise.

Contexts are expressions with a hole defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$. We will mostly use *pattern-substitutions* $PSubst = \{\theta \in Subst \mid \theta(X) \in Pat, \forall X \in \mathcal{V}\}$.

A *HOCRWL*-program (or simply a *program*) consists of one or more *program rules* for each $f \in FS^n$, having the form $f\ t_1 \ldots t_n \to r$ where $(t_1, \ldots, t_n)$ is a linear (i.e. variables occur only once) tuple of (maybe HO) patterns and r is any expression. Notice that confluence or termination is not required, and that r may have variables not occurring in

$f\ t_1 \ldots t_n$, i.e., extra variables are allowed in program rules (we write $vExtra(R)$ for such variables in a rule R).

Some related languages, like *Curry*, do not allow HO-patterns in left-hand sides of function definitions. We remark that all the notions and results in the paper are applicable to programs with this restriction and we stress the fact that Ex. 3.2.7 is one of them.

The *HOCRWL* proof calculus The semantics of a program $\mathcal{P}$ is determined in *HOCRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\perp$ and $t \in Pat_\perp$, meaning informally that t is (or approximates to) a possible value of e, obtained by evaluation of e using $\mathcal{P}$ under call-time choice.

The *HOCRWL*-proof calculus is presented in Fig. 3.13. We write $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in that calculus using the program $\mathcal{P}$. The *HOCRWL-denotation* of an expression $e \in Exp_\perp$ is defined as $[\![e]\!]^{\mathcal{P}}_{HOCRWL} = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t\}$. $\mathcal{P}$ and *HOCRWL* are frequently omitted in those notations. Note that, as it happened with the original first order version of *CRWL*, *HOCRWL*-denotations are possibly infinite sets of finite partial approximations to possibly infinite (higher order, in this case) values.

$$\textbf{B}\ \frac{}{e \twoheadrightarrow \perp} \qquad \textbf{RR}\ \frac{}{x \twoheadrightarrow x}\ x \in \mathcal{V}$$

$$\textbf{DC}\ \frac{e_1 \twoheadrightarrow t_1\ \ldots\ e_n \twoheadrightarrow t_m}{h\ e_1 \ldots e_m \twoheadrightarrow h\ t_1 \ldots t_m}\quad h \in \Sigma,\ \text{if } h\ t_1 \ldots t_m \text{ is a partial pattern, } m \geq 0$$

$$\textbf{OR}\ \frac{e_1 \twoheadrightarrow t_1 \ldots\ e_n \twoheadrightarrow t_n\ \ r\ a_1 \ldots a_m \twoheadrightarrow t}{f\ e_1 \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t}\quad \text{if } m \geq 0, (f\ t_1 \ldots t_n \to r) \in [\mathcal{P}]_\perp$$

Figure 3.13: *HOCRWL*-calculus

In Ex. 3.2.7 we have $[\![f double\ f\ 0]\!] = \{0, s\ (s\ 0), \perp, s\ \perp, s\ (s\ \perp)\}$ and $[\![f double\ f'\ 0]\!] = \{0, s\ 0, s\ (s\ 0), \perp, s\ \perp, s\ (s\ \perp)\}$.

The following simplification of Th. 1 from [LRS08a] states an important compositionality property of the semantics of *HOCRWL*-expressions: the semantics of a whole expression depends only on the semantics of its constituents, in a particular form reflecting the idea of call-time choice.

Theorem 3.2.15 (Compositionality of *HOCRWL* semantics, [LRS08a] Th. 1)**.**

$$[\![\mathcal{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathcal{C}[t]]\!]$$

for any program $\mathcal{P}$ *and expression* $e \in Exp_\perp$.

HOLet-rewriting

We have just introduced the extension to higher order of the *CRWL* framework, now we will present the extentions to HO of the notions of *let* rewriting and narrowing, first developed in [LRS08a] (Sect. 7.1.2, page 126).

Syntax As we did in the first order version, to express sharing, as is required for call-time choice, we enhance the syntax of expressions (and contexts) with a *let* construct for local bindings, in the spirit of [AFM+95, MOW98, LRS07b]:

$$\begin{array}{lcl} LExp \ni e & ::= & X \mid h \mid e_1\ e_2 \mid let\ X = e_1\ in\ e_2 \\ Cntxt \ni \mathcal{C} & ::= & [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \end{array}$$

The notion of *shell* of an expression has also to be adapted to the higher order setting. Written as $|e|$, is a pattern containing the 'stable' outer information of e, not to be destroyed by reduction:

$$|X\ e_1 \ldots e_m| = \begin{cases} X & \text{if } m = 0 \\ \bot & \text{if } m > 0 \end{cases}$$

$$|h\ e_1 \ldots e_m| = \begin{cases} h\ |e_1| \ldots |e_m| & \text{if } (h \in CS^n, m \leq n) \text{ or } (h \in FS^n, m < n) \\ \bot & \text{otherwise (junk or active expression)} \end{cases}$$

$$|(let\ X = e_1\ in\ e_2)\ a_1 \ldots a_m| = |(e_2[X/e_1])\ a_1 \ldots a_m|$$

Notice that in FO—see Sect. 3.2.2 (page 38)— we defined $|(let\ X = e_1\ in\ e_2)| = |e_2|[X/|e_1|]$. This would loose information in the HO case: for instance, $|let\ X = s\ in\ X\ 0|$ would be $\bot$, instead of the more accurate $s\ 0$ given by the definition above. Anyway it is easy to check that in first order $|e_2|[X/|e_1|] \equiv |(e_2[X/e_1])|$, for any $e_1, e_2 \in LExp$.

As it happened with its first order version, the *HOlet*-rewriting relation is able to handle extra variables, thus it works with *HOCRWL*-programs with extra variables as those defined in Sect. 3.2.4 (page 50).

Rules of *HOLet*-rewriting Figure 3.14 defines the *HOlet*-rewriting relation $\rightarrow^l$.

Fapp $f\ t_1 \ldots t_n \rightarrow^l r$, if $(f\ t_1 \ldots t_n \rightarrow r) \in [\mathcal{P}]$

LetIn $e_1\ e_2 \rightarrow^l let\ X = e_2\ in\ e_1\ X$ (X fresh), if e_2 is an active expression, variable application, junk or *let* rooted expression.

Bind $let\ X = t\ in\ e \rightarrow^l e[X/t]$, if $t \in Pat$

Elim $let\ X = e_1\ in\ e_2 \rightarrow^l e_2$, if $X \notin FV(e_2)$

Flat $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

LetAp $(let\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let\ X = e_1\ in\ e_2\ e_3$, if $X \notin FV(e_3)$

Contx $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrow^l e'$ using any of the previous rules, and in case $e \rightarrow^l e'$ is a (Fapp) step using $(f\ \overline{p} \rightarrow r)\theta \in [\mathcal{P}]$ then $vran(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$.

Figure 3.14: Higher order *let*-rewriting relation $\rightarrow^l$ [LRS08a]

Rule Fapp uses a program rule to reduce a function application, but only when the arguments are already patterns, otherwise call-time choice would be violated. Non-pattern arguments of applications are moved to local bindings by LetIn. Local bindings of patterns to variables are applied in Bind, since in this case copying is harmless. Elim erases useless bindings. Flat and LetAp manage local bindings; they are needed to avoid some reductions to get stuck. Finally, any of these rules can be applied to any subexpression by

CONTX. It includes an additional technical condition to avoid undesired variable captures when FAPP was applied inside a surrounding context and the used program rule has extra variables. If, for instance, a program rule is $f \to Y$, the rule CONTXT avoids the step $let\ X{=}0\ in\ f \to^l let\ X{=}0\ in\ X$ and also the step $let\ X{=}f\ in\ X \to^l let\ X{=}X\ in\ X$.

Example 3.2.8. The following derivation corresponds to Example 3.2.7 (page 49):

$$\begin{array}{ll}
fdouble\ f\ 0 \to^l (let\ F = f\ in\ fdouble\ F)\ 0 & \text{LETIN, CNTXT} \\
\to^l let\ F = f\ in\ fdouble\ F\ 0 & \text{LETAP} \\
\to^l let\ F = f\ in\ fadd\ F\ F\ 0 & \text{FAPP, CNTXT} \\
\to^l let\ F = g\ in\ fadd\ F\ F\ 0 & \text{FAPP, CNTXT} \\
\to^l g\ 0 + g\ 0 & \text{BIND} \\
\to^{l^*} 0 &
\end{array}$$

Notice that the first step is justified because f is active. In contrast, since f' is a pattern, a derivation for *fdouble f' 0* could proceed as follows:

$$\begin{array}{ll}
fdouble\ f'\ 0 \to^l fadd\ f'\ f'\ 0 & \text{FAPP} \\
\to^l f'\ 0 + f'\ 0 & \text{FAPP} \\
\to^{l^*} f\ 0 + f\ 0 \to^{l^*} g\ 0 + h\ 0 \to^{l^*} s\ 0 &
\end{array}$$

Adequacy of *HOLet*-rewriting Again as we did in first order, the first thing we have to do is extending the *HOCRWL* calculus to cope with *lets*. The $HOCRWL_{let}$ proof calculus proves statements $e \twoheadrightarrow t$ ($e \in LExp_\bot, t \in Pat_\bot$), and results from adding to Fig. 3.13 the rule:

$$\textbf{LET}\quad \frac{e_1 \twoheadrightarrow t_1 \quad (e_2[X/t_1])\ a_1 \ldots a_m \twoheadrightarrow t}{(let\ X = e_1\ in\ e_2)\ a_1 \ldots a_m \twoheadrightarrow t} \qquad (m \geq 0)$$

Theorem 3.2.15 does not hold as it is for *let*-expressions (assume, for instance, the program rule $f\ 0 = 1$ and take $e \equiv f\ X$, $\mathcal{C} \equiv let\ X{=}0\ in\ [\,]$). However, a more limited form of compositionality is still available.

Theorem 3.2.16 (Weak compositionality of $HOCRWL_{let}$ semantics, [LRS08a] Th. 2)**.** *For any $\mathcal{P}$ and $e, e' \in LExp_\bot$: $[\![C\ [e]]\!] = \bigcup_{t \in [\![e]\!]} [\![C\ [t]]\!]$, if $BV(C) \cap FV(e) = \emptyset$.*

As in first order *let*-rewriting, a HO version of hypersemantics is defined to cope with the lack of monotonicity under context of *HOCRWL* denotations. Using this hypersemantics we get our first soundness result.

Lemma 3.2.4 (One-Step Hyper-Soundness of *HOlet*-rewriting, [LRS08a] Lemma 2)**.** *$e \to^l e'$ implies $[\![e']\!] \Subset [\![e]\!]$, for any $e, e' \in LExp$.*

Notice that $\Subset$ cannot be replaced here by $=$, due to non-determinism. Lemma 3.2.4, together with the easy observation that $[\![e_1]\!] \Subset [\![e_2]\!]$ implies $[\![e_1]\!] \subseteq [\![e_2]\!]$ (just take $\theta = \epsilon$) and an obvious induction over derivation lengths, leads to our main correctness result for $\to^l$:

Theorem 3.2.17 (Soundness of *HOlet*-rewriting, [LRS08a] Th. 3)**.** *Let $\mathcal{P}$ be a program, $e, e' \in LExp$. Then: (i) $e \to^{l^*} e'$ implies $[\![e']\!] \subseteq [\![e]\!]$, and therefore $e \twoheadrightarrow |e'|$*
(ii) $e \to^{l^*} t$ implies $e \twoheadrightarrow t$, for any $t \in Pat$.

Regarding completeness, it is achieved by an adaptation to HO of the techniques used in the original first order version of *let*-rewriting, getting the following result.

Theorem 3.2.18 (Completeness of *HOlet*-rewriting, [LRS08a] Th. 4)**.** *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in Pat_\perp$. Then:*
(i) $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ *implies* $e \rightarrow^{l*} e'$, *for some* $e' \in LExp$ *such that* $t \sqsubseteq |e'|$.
(ii) If in addition $t \in Pat$, *then* $e \rightarrow^{l*} t$.

Joining together the last parts of Theorems 3.2.17 and 3.2.18, we obtain a strong equivalence result for $\rightarrow^l$ and $\twoheadrightarrow$:

Theorem 3.2.19 (Equivalence of *HOlet*-rewriting and *HOCRWL*, [LRS08a] Th. 5)**.**
$\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ *iff* $e \rightarrow^{l*} t$, *for any* $\mathcal{P}$, $e \in Exp$, *and* $t \in Pat$.

This justifies our claim that $\rightarrow^l$ is truly the reduction face of *HOCRWL*-semantics.

HOLet-narrowing

In Sect. 3.2.3 we proposed a notion of narrowing adequate to FO *let*-rewriting, and now we extend it to HO. As happens in [GHR97, AT99], *HOlet*-narrowing may bind variables to HO-patterns. Hence, like in the FO version, variables bound by *let* bindings must be protected against bindings generated by narrowing, for the same reasons argued in the definition of FO *let*-rewriting: the role of *let*-bound variables is to express intermediate values that are evaluated at most once and shared, and anyway narrowing *let*-bound variables may destroy the structure of let-expressions. But here we will take a different approach to protect these bound variables. Instead of considering a distinguished set $PVar \subset \mathcal{V}$ of produced variables and require conditions to the rules of the calculus regarding this special variables, we will consider a unique set of variables $\mathcal{V}$ and require the same conditions on the Contx steps, to variables bound by *let* bindings in the context used in that Contx step. That is, instead of imposing local conditions we will impose a global condition in the rule that gives us that global view: the Contx rule. Although this approach complicates some proofs (see the formulation of Lemma 3.2.5, for example), we think that the overall approach is much cleaner, and this extra effort well worth it.

Figure 3.15 contains the rules for the one-step *HOlet*-narrowing relation $e \leadsto^l_\theta e'$, expressing that e is narrowed to e' producing the substitution $\theta \in PSubst$. In X we collect those cases of *HOlet*-rewriting corresponding also to narrowing steps with empty substitution. Narr is the proper rule of narrowing for function application; it may produce HO bindings if the used program rule has HO patterns. Notice that, for the sake of generality, we do not require that θ is a mgu. VAct and VBind are rules producing HO bindings for flexible expressions (or subexpressions, in the case of VBind). We have preferred this pair of rules instead of the rule

$$\textbf{VNarr} \quad X\ e \leadsto^l_{[X/t]} t\ (e[X/t]), \text{ for any } t \in Pat$$

which is simpler, but also 'wilder' because it creates a larger search space. Finally, Contxt is a contextual rule where, as in first order *let*-narrowing , it is crucial to protect bound variables from narrowing (condition (i)) and to avoid variable capture (condition (ii), automatically fulfilled if mgu's are used in Narr and VAct, and fresh *shallow* patterns –i.e., of the form $h\ X_1 \ldots X_n$– in VBind).

X $e \leadsto^{l}_{\epsilon} e'$ if $e \to^{l} e'$ using $\boldsymbol{X} \in \{\text{ELIM}, \text{BIND}, \text{FLAT}, \text{LETIN}, \text{LETAP}\}$ in Figure 3.14.

NARR $f\ \overline{t} \leadsto^{l}_{\theta} r\theta$, for any fresh variant $(f\ \overline{p} \to r) \in \mathcal{P}$ and $\theta \in PSubst$ such that $f\ \overline{t}\theta \equiv f\ \overline{p}\theta$.

VACT $X\ t_1 \ldots t_k \leadsto^{l}_{\theta} r\theta$, if $k > 0$, for any fresh variant $(f\ \overline{p} \to r) \in \mathcal{P}$ and $\theta \in PSubst$ such that $(X\ t_1 \ldots t_k)\theta \equiv f\ \overline{p}\theta$.

VBIND $let\ X = e_1\ in\ e_2 \leadsto^{l}_{\theta} e_2\theta[X/e_1\theta]$, if $e_1 \notin Pat$, for any $\theta \in PSubst$ that makes $e_1\theta \in Pat$, provided that $X \notin (dom(\theta) \cup vran(\theta))$.

CONTX $\mathcal{C}[e] \leadsto^{l}_{\theta} \mathcal{C}\theta[e']$ for $\mathcal{C} \neq [\,]$, if $e \leadsto^{l}_{\theta} e'$ by any of the previous rules, and the following conditions hold:

i) $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$

ii) • If the step is NARR or VACT using $(f\ \overline{p} \to r) \in \mathcal{P}$, then $vran(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$
• If the step is VBIND then $vran(\theta) \cap BV(\mathcal{C}) = \emptyset$

Figure 3.15: Higher order *let*-narrowing calculus $\leadsto^{l}$ [LRS08a]

Example 3.2.9. Taking Example 3.2.7 (page 49), a narrowing derivation for $fdouble\ F\ 0$ would start with some X 'rewriting' steps:

$$fdouble\ F\ 0 \leadsto^{l}_{\epsilon} fadd\ F\ F\ 0 \leadsto^{l}_{\epsilon} F\ 0 + F\ 0 \leadsto^{l}_{\epsilon} let\ X = F\ 0\ in\ X + F\ 0$$

At this point, notice first that we cannot narrow on X, because it is a bound variable. Instead, we can apply VACT+CONTX:

$$let\ X = F\ 0\ in\ X + F\ 0 \leadsto^{l}_{\{F/g\}} let\ X = 0\ in\ X + g\ 0 \leadsto^{l*}_{\epsilon} 0$$

Other similar derivations using VACT+CONTX would bind F to h (with final result $s\ (s\ 0)$), or to f' (with possible results $0, s\ 0, s\ (s\ 0)$). Notice that the binding X/f is not legal, since f is not a pattern. Alternatively we could have applied VBIND, obtaining:

$$let\ X = F\ 0\ in\ X + F\ 0 \leadsto^{l}_{\{F/s\}} s\ 0 + s\ 0 \leadsto^{l*}_{\epsilon} s\ (s\ 0)$$

We remark that, in our untyped framework, other 'ill-typed' bindings could be tried, like $F/fadd\ 0$ or $F/fdouble$. This is a symptom of known problems [AT99, GHR01] of the interaction with types of the intensional view of HO, that are partially alleviated in [AT99] by a typed version of a FO translation, but in general require (see [GHR01]) bringing types to computations, a problem yet not well solved in practice. All these type-related issues are out of the scope of this work, although we have made some advances in [LMR10].

Adequacy of *HOLet*-narrowing Proving the adequacy of *HOLet*-narrowing wrt. *HOLet*-rewriting becomes a more complicated task that in the first order setting, because some important properties of reductions do not hold anymore. We refer the reader to [LRS08a] (Sect. 7.1.2, page 126) for details about the additional relations and results needed to prove the following soundness results for *HOLet*-narrowing.

Theorem 3.2.20 (**Soundness** of ***HOlet*-narrowing**, [LRS08a] Th. 7)**.** *For any* $e, e' \in LExp$, $t \in Pat$*:*

a) If $e \leadsto^{l*}_{\theta} e'$ *then* $[\![e']\!] \subseteq [\![e\theta]\!]$

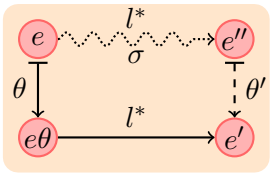
b) If $e \leadsto^{l*}_{\theta} t$ *then* $e\theta \to^{l*} t$

Regarding completeness, the key result is again an adaptation of Hullot's *lifting lemma* [Hul80] to the framework of *HOlet*-rewriting, that shows how we can lift any $\to^l$ derivation to a $\leadsto^l$ derivation.

Lemma 3.2.5 (Lifting lemma for *HOlet*-rewriting, [LRS08a] Lemma 6)**.** *Let* $e, e' \in LExp$ *such that* $e\theta \to^{l*} e'$ *for some* $\theta \in PSubst$*, and let* $\mathcal{W}, \mathcal{B} \subseteq \mathcal{V}$ *with* $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$*,* $BV(e) \subseteq \mathcal{B}$ *and* $(dom(\theta) \cup vran(\theta)) \cap \mathcal{B} = \emptyset$*, and for each instance of a program rule* $R\gamma \in [\mathcal{P}]$ *used in an* FApp *step of* $e\theta \to^{l*} e'$ *then* $vran(\gamma|_{vExtra(R)}) \cap \mathcal{B} = \emptyset$*. Then there exist a derivation* $e \leadsto^{l*}_{\sigma} e''$ *and* $\theta' \in PSubst$ *such that:*

$$\textit{(i)}\ e''\theta' = e' \qquad \textit{(ii)}\ \sigma\theta' = \theta[\mathcal{W}] \qquad \textit{(iii)}\ (dom(\theta') \cup vran(\theta')) \cap \mathcal{B} = \emptyset$$

Besides, the HOlet*-narrowing derivation can be chosen to use mgu's at each* Narr *or* VAct *step, and fresh shallow patterns in the range for each* VBind *step. Graphically:*



Moreover in every Narr *step the mgu was used.*

With the aid of this lemma we can reach our completeness result for $\leadsto^l$:

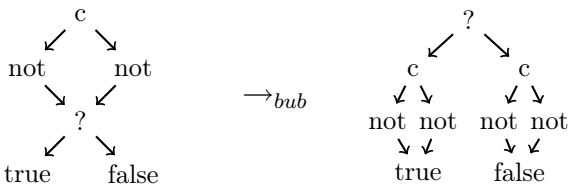
Theorem 3.2.21 (Completeness of *HOlet*-narrowing wrt. *HOlet*-rewriting, [LRS08a] Th. 8)**.** *Let* $e, e' \in LExp$ *and* $\theta \in PSubst$*. If* $e\theta \to^{l*} e'$*, then there exist a* HOlet*-narrowing derivation* $e \leadsto^{l*}_{\sigma} e''$ *and* $\theta' \in PSubst$ *such that* $e''\theta' \equiv e'$ *and* $\sigma\theta' = \theta[FV(e)]$*.*

Some applications of the framework

In [LRS08a] we can find two sample applications of the framework of *HOlet*-rewriting and how its powerful combination with *HOCRWL* simplifies the reasoning about the meaning and behaviour of FLP programs. The point is that having equivalent notions of semantics and reduction allows to reason interchangeably at the rewriting and the semantic levels.

Bubbling Bubbling, proposed in [ABC07], is an operational rule devised to improve the efficiency of functional logic computations. Its correctness was formally studied in [ABC06] in the framework of the variant [EJ97, EJ98] of term graph rewriting (see Sect. 3.1.4).

The idea of bubbling is to concentrate all non-determinism of a system into a *choice* operation ? defined by the rules $X\ ?\ Y \to X$ *and* $X\ ?\ Y \to Y$, and to lift applications of ? out of a surrounding context, as illustrated by the following graph transformation taken from [ABC06]:



As it is shown in [ABC07], bubbling can be implemented in such a way that many functional logic programs become more efficient, but we will not deal with these issues here.

Due to the technical particularities of term graph rewriting, not only the proof of correctness, but even the definition of bubbling in [ABC07, ABC06] are involved and need subtle care concerning the appropriate contexts over which choices can be bubbled. In contrast, bubbling can be expressed within our framework (moreover, generalized to HO) in a remarkably easy and abstract way as a new rewriting rule, in Fig. 3.16.

Bub $\quad \mathcal{C}[e_1?e_2] \to^{bub} \mathcal{C}[e_1]?\mathcal{C}[e_2]$

Figure 3.16: The Bubbling Rule [LRS08a]

The fact that bubbling preserves $HOCRWL_{let}$-semantics has a simple formulation:

Theorem 3.2.22 (Correctness of bubbling, [LRS08a] Th. 9)**.** *If* $e \to^{bub} e'$*, then* $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$*. In other terms,* $[\![\mathcal{C}[e_1?e_2]]\!] = [\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!]$ $(= [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!])$*, for any* $e_1, e_2 \in$ *LExp and context* $\mathcal{C}$*.*

From this and the adequacy of *HOlet*-rewriting we obtain as immediate corollary the correctness of bubbling in terms of rewriting:

Corollary 3.2.3 ([LRS08a] Corollary 1)**.** $e \to_l^* t \Leftrightarrow e\ (\to_l \cup \to_{bub})^*\ t$

It is interesting to observe that most of the proof of Th. 3.2.22 consists of direct calculations with denotation of expressions, in the form of chains of equalities of denotations, justified by general properties of the semantics like the compositionality of the semantics. We find this methodology quite appealing and refer the reader to [LRS08a] (Sect. 7.1.2, page 126) for details.

Translation to first order Since [War82], a common technique to implement HO features in FO settings consists in a *HO-to-FO* translation introducing data constructors to represent partial applications and a special function @ (read *apply*) for reducing application of such constructors. This has been used within the context of *FLP* in [Gon93, AT99]. We have adapted such a transformation to our context and provided a correctness proof with respect to the semantics of the source and object programs, given by *HOCRWL* and *CRWL* respectively.

In [LRS08a], given a HO program $\mathcal{P}$, a transformation to obtain the corresponding FO program $\mathcal{P}_{fo}$ is defined, as well as a transformation *fo* from HO expressions to FO expressions. The main result about the adequacy of this transformation is the following.

Theorem 3.2.23 (Adequacy of HO-to-FO translation, [LRS08a] Th. 10)**.** *Let* $\mathcal{P}$ *be a program,* $e \in Exp_\bot$*,* $t \in Pat_\bot$*. Then:* $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t \Leftrightarrow \mathcal{P}_{fo} \vdash_{CRWL} fo(e) \twoheadrightarrow fo(t)\!\downarrow_@$
Or, in terms of HOlet*-rewriting:* $e \to^{l^*} t \Leftrightarrow fo(e) \to^{l^*} fo(t)\!\downarrow_@$

This result is relevant, not only because this transformational technique is actually used in the implementations of FLP systems, but also because this is the first formal proof that it still behaves properly when non-deterministic functions with call-time choice are considered, as previous works in the field of FLP [Gon93, AT99] consider only deterministic functions. More details about the transformation and the proof for its adequacy can be found in [LRS08a] (Sect. 7.1.2, page 126).

HOLet **Rewriting and Narrowing: conclusions**

We have addressed the broad question: *what means 'reduction' for functional logic programming?*, which had no previous satisfactory answer for the combination *HO + non-deterministic functions + call-time choice* supported by current systems in the mainstream of the field like *Toy* or *Curry*. To do that we have based on the declarative semantics of [GHR97], an extension to higher order of *CRWL* called *HOCRWL*, and use it to adapt our *let*-rewriting and *let*-narrowing relations to a HO context.

A number of relevant technical results have been obtained along the way. We have proved the equivalence of *HOlet*-rewriting wrt. to *HOCRWL*, established new compositional properties for *HOCRWL* and extended this logic to cope with *let*s. Then, after lifting *HOlet*-rewriting to a notion of *HOlet*-narrowing, the soundness and completeness of this new narrowing notion have been proved. We have also tried to apply our framework in some concrete scenarios, using the problems of the correctness of bubbling [ABC07] and the translation to first order of Warren [War82] as two interesting sample applications.

Adapting the first order versions of *let* rewriting and narrowing to the higher order case has not been a routine task at all; on the contrary, some results have been indeed a technical challenge, because of the subtle problems caused by the use of a call-time choice semantics in a higher order environment, as seen in Ex. 3.2.7 (page 49).

Our wish with this work—jointly with the one developed in sections 3.2.2 and 3.2.3 defining the first order versions of *let* rewriting and narrowing—is to provide foundational pieces useful to understand how a FLP computation proceeds, serving also as suitable technical basis to address in the call-time choice context other operational issues (rewriting and narrowing strategies, residuation, program optimization, types in computations,...), all of which are lines of future work. Among those, we are particularly interested in the problems caused by adoption in FLP systems like *Toy* or *Curry* of a Damas & Milner type system [DM82]. The point is that this type system is devised to work with ground expressions and programs without extra variables—as it is usual in functional languages, where a free variable is understood as an 'unknown identifier'—, while FLP systems use programs with extra variables and handle expressions with free variables which get bound by narrowing. A precise descriptions of those problems can be found in [GHR01]. We have already made some progress in that direction in [LMR10], where we used *HOlet*-rewriting as the formulation of the operational behaviour of FLP systems.

3.2.5 The Full Abstraction Problem for Higher Order FLP

In this section we will show the results concerning full abstraction of higher order FLP programs presented in [LR10] (Sect. 7.2.5, page 151). *Full abstraction* was introduced by Plotkin [Plo77] in connection to PCF, a simple model of functional programming based on λ-calculus. It is a highly desirable property, indicating a perfect correspondence between the semantics and the behavior of program pieces, according to a given criterion of *observation*. We say that a semantics is fully abstract when for any pair of expressions these have the same semantics iff they are observationally indistinguishable when put in any context.

Regarding full abstraction in PCF, Plotkin realized that the standard Scott semantics, in which expressions of functional types have classical mathematical functions as meanings, lacks full abstraction with respect to observing the value obtained in the evaluation of an

expression. The reason lays in the impossibility of defining the function *por* (*parallel or*) in PCF. Using this fact one can build two higher order (HO) expressions e_1, e_2 denoting two different mathematical functions φ_1, φ_2, both expecting boolean functions as arguments, such that φ_1, φ_2 only differ when applied to *por* as argument. Therefore e_1, e_2 have different Scott semantics but this difference cannot be *observed*. It is usually said that the semantics is *too concrete*. Notice, however, that Scott semantics for PCF is *sound*, that is, if two expressions have the same semantics, they cannot be observably distinguished. Unsoundness of a semantics can be considered a flaw, much more severe that being too concrete, which is more a weakness than a flaw.

Full abstraction for PCF was achieved in different technical ways (see e.g. [BCL86]). But for our purposes it is more interesting to recall that the Scott semantics becomes fully abstract if PCF is enriched with the 'missing' *por* function (see e.g. [Mit96]). As we have seen in previous sections, the mainstream of functional logic programming uses constructor systems as programs, and as a consequence parallel or can be defined straightforwardly by an overlapping (almost orthogonal) rewriting system. So one could think of assigning to FLP languages a denotational semantics in the FP style. For instance, for a definition like $f\ x = 0$, one could assign to f the meaning $\lambda x.0$. But as FLP functions can be non-deterministic, the standard semantics should be modified in the sense that the meaning of a function would be some kind of set-valued function.

Nevertheless, *this roadmap cannot be followed anymore when non-determinism is combined with HO and call-time choice*, which as we know is the notion of non-determinism adopted in modem implementations of FLP like *Toy* or *Curry*. Ex. 3.2.7 ([LRS08a] Ex. 1, [LR10] Ex. 1) already shows the problem. We reproduce here the program of Ex. 3.2.7 for the sake of convenience.

```
g X -> 0          f -> g          f' X -> f X
h X -> s 0        f -> h

fadd F G X -> (F X) + (G X)       fdouble F -> fadd F F
```

Here f and f' are non-deterministic functions that are (by definition of f') extensionally equivalent. In a set-valued variant of Scott semantics, their common denotation would be the function $\lambda X.\{0, s\ 0\}$, or something essentially equivalent. But this leads to unsoundness of the semantics. The point is that, as seen in Ex. 3.2.7, *fdouble f' 0* can be reduced to *s 0* while it is not the case for *fdouble f 0*. We see then that f and f' can be put in a context able to distinguish them, implying that any semantics assigning f and f' the same denotation is necessarily unsound, and therefore not fully abstract.

On the other hand, as we saw in previous sections, *HOCRWL* adopts an *intensional* view of functions, where different descriptions –in the form of *HO-patterns*– of the same extensional function are distinguished as different data. Therefore it remains an open question whether full abstraction for FLP can be achieved by using *HOCRWL* to define the semantics of expressions or not. That question will be elucidated in the present section. As we will get positive answers —at least for programs without extra variables—, an anticipated conclusion of our work is that one must take into account intensional descriptions of functions as sensible meanings of expressions in HO non-deterministic FLP programs, even if one does not want to explicitly program with HO-patterns.

The Full Abstraction Problem for FLP

Full abstraction depends on a criterion of observability for expressions. In constructor based languages, like FLP languages, it is reasonable to observe the outcomes of computations, given by constructor forms reached by reduction. Here, we can interpret 'constructor form' in a liberal sense, including HO-patterns, or in a more restricted sense, only with FO-patterns. This leads to the following notions of observation.

Definition 3.2.3 (observations, [LR10] Def. 2)**.** Let $\mathcal{P}$ be a program. We consider the following observations:

- $\mathcal{O}^{\mathcal{P}} : Expr \mapsto Pat$ is defined as $\mathcal{O}^{\mathcal{P}}(e) = \{t \in Pat \mid \mathcal{P} \vdash e \to^{l^*} t\}$
- $\mathcal{O}^{\mathcal{P}}_{fo} : Expr \mapsto FOPat$ is defined as $\mathcal{O}^{\mathcal{P}}_{fo}(e) = \{t \in FOPat \mid \mathcal{P} \vdash e \to^{l^*} t\} (= \mathcal{O}^{\mathcal{P}}(e) \cap FOPat)$

Now we turn to the definition of full abstraction. In programming languages like PCF the condition for full abstraction is usually stated as:

$$[\![e]\!] = [\![e']\!] \Leftrightarrow \mathcal{O}(\mathcal{C}[e]) = \mathcal{O}(\mathcal{C}[e']), \text{for any context } \mathcal{C} \tag{3.1}$$

where $\mathcal{O}$ is the observation function of interest. Programs do not need to be mentioned, because programs and expressions can be identified by contemplating the evaluation of e under $\mathcal{P}$ as the evaluation of a big λ-expression or big *let*-expression embodying $\mathcal{P}$ and e. Contexts pose no problems either. In our case, since programs are kept different from expressions, some care must be taken. It might happen that $\mathcal{P}$ has not enough syntactical elements and rules to built interesting distinguishing contexts. For instance, if in Ex. 3.2.7 we drop the definition of $fdouble$, and we consider $\mathcal{O}_{fo}$ as observation, then we cannot built a context that distinguishes f from f'. This would imply that soundness or full abstraction would not be intrinsic to the semantics, but would greatly depend on the program. What we need is requiring the right part of (3.1) to hold for all contexts that might be obtained by extending $\mathcal{P}$ with new auxiliary functions. To be more precise, we say that $\mathcal{P}'$ is a *safe extension* of $(\mathcal{P}, e)$ if $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}''$, where $\mathcal{P}''$ does not include defining rules for any function symbol occurring in $\mathcal{P}$ or e. The following property of *HOCRWL* regarding safe extensions will be crucial for full abstraction. The property is subtler than it appears to be, as witnessed by the fact that it fails to hold if programs have extra variables, as discussed later.

Lemma 3.2.6 ([LR10] Lemma 1)**.** *For any program $\mathcal{P}$ without extra variables, $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$ when $\mathcal{P}'$ safely extends (P, e).*

With the aid of the notion of safe extension we can now formalize the full abstraction problem as follows.

Definition 3.2.4 (Full abstraction, [LR10] Def. 3)**.**

a) A semantics is *fully abstract* wrt. $\mathcal{O}$ iff for any $\mathcal{P}$ and $e, e' \in Expr$, the following two conditions are equivalent:

   i) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$

   ii) $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$ for any $\mathcal{P}'$ safely extending $(\mathcal{P}, e)$, $(\mathcal{P}, e')$ and any $\mathcal{C}$ built with the signature of $\mathcal{P}'$.

In words: semantic equality is equivalent to observational indistinguishability.

b) A notion weaker than full abstraction is: a semantics is *sound* wrt. $\mathcal{O}$ iff the condition *i)* above implies the condition *ii)*. In words: semantic equality implies observational indistinguishability.

iii) A semantics is *compositional* iff for any $\mathcal{P}$ and $e, e' \in Expr$, the following two conditions are equivalent:

a) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$

b) $[\![\mathcal{C}[e]]\!]^{\mathcal{P}} = [\![\mathcal{C}[e']]\!]^{\mathcal{P}}$ for any $\mathcal{C}$ built with the signature of $\mathcal{P}$.

In words: the semantics of an expression depends only on the semantics of its subexpressions. Notice that $b) \Rightarrow a)$ holds trivially (take $\mathcal{C} = [\ ]$).

For extensional semantics the full abstraction property does not hold. For example the following is a sensible definition for a family of extensional semantics for FLP.

Definition 3.2.5 (See [LR10] Def. 1)**.** Given $n \geq 0$, $e \in Expr_\bot$, the n-extensional semantics of e is defined as: $[\![e]\!]_{ext_n} = \lambda t_1 \ldots \lambda t_n.\ [\![e\ t_1 \ldots t_n]\!]$ $(t_i \in Pat_\bot)$.

As seen above, Ex. 3.2.7 (and obvious generalizations to arities $k > 1$) constitutes a proof of the following negative result for extensional semantics

Proposition 3.2.2 ([LR10] Prop. 2)**.** For any $k > 0$, $[\![\_]\!]_{ext_k}$ is not compositional nor sound nor full abstract wrt. $\mathcal{O}, \mathcal{O}_{fo}$. This remains true even if programs are restricted to be left-FO, that is, when the use of HO-patterns in left-hand sides of program rules is forbidden.

This contrast with the following:

Theorem 3.2.24. *$[\![\_]\!]$ is a compositional semantics in the sense of Def. 3.2.4.*

Proof. A straightforward corollary of Th. 3.2.15 (page 51). □

Theorem 3.2.25 (Full abstraction, [LR10] Th. 3)**.** *When restricted to programs without extra variables, $[\![\_]\!]$ is fully abstract wrt. $\mathcal{O}$ and $\mathcal{O}_{fo}$.*

Discussion: the case of extra variables

In the results above we have assumed the absence of extra variables in program rules, i.e., that $var(r) \subseteq var(f\ t_1 \ldots t_n)$ holds for any program rule $f\ t_1 \ldots t_n \to r$. This condition is necessary for the full abstraction results to hold, as discussed in detail in [LR10] (Sect. 7.2.5, page 151). The point is that, if extra variables are allowed, Lemma 3.2.6 does not hold in general anymore, which in fact is what it is used in our proofs for full abstraction—besides compositionality—, and what is exploited in the (counter-)examples shown in [LR10]. We contemplate the extension of this work to cope with extra variables as a challenging subject of future work.

The Full Abstraction Problem for Higher Order FLP: conclusions

We have seen that reasoning extensionally in existing FLP languages with HO nondeterministic functions is not valid in general (Ex. 3.2.7, Prop. 3.2.2). In contrast, thinking in intensional functions is not an arbitrary exoticism, but rather an appropriate point of view for that setting (Th. 3.3.4). We stress the fact that adopting an intensional view of the *meaning* of functions is compatible with a discipline of programming in which programs are restricted to be left-FO, that is, the use of HO-patterns in left-hand sides of program rules is forbidden. This is the preferred choice by some people in the FLP community, mostly because HO-patterns in left-hand sides cause some problems to the type system—a precise description of this problematic aspects, as well as a solution for some of these problems can be found in [GHR01, LMR10].

We have also seen how the presence of extra variables in programs destroys full-abstraction of the *HOCRWL* semantics. Recovering it for such family of programs is an obvious subject of future work. Another very interesting, and somehow related matter, is giving variables a more active role in the semantics. Certainly, the results in the paper are not restricted to ground expressions, but their interest for expressions having variables is limited by the fact that in the notions of semantics and observations considered in the paper, variables are implicitly treated as generic constants. For instance, the expressions $e_1 \equiv X + X$ and $e_2 \equiv X + 0$ do have the same semantics $[\![\_]\!]_\bot$ ($[\![e_1]\!]_\bot = [\![e_2]\!]_\bot = \{\bot\}$). Full abstraction of $[\![\_]\!]_\bot$ ensure that $\mathcal{O}(\mathcal{C}[e_1]) = \mathcal{O}(\mathcal{C}[e_2])$ for any context $\mathcal{C}$. This is ok as far as one is only interested in possible reductions starting from e_1, e_2. If this is the case, certainly e_1 and e_2 have equivalent behavior (no successful reduction to a pattern can be done with any of them). However, in some sense e_1 and e_2 have different 'meanings', that are reflected in different behaviors; for instance, if e_1 and e_2 are subject to narrowing, or if e_1 and e_2 are used as right hand sides in a program rule.

3.2.6 Reasoning about *CRWL* in Isabelle/Isar/HOL

In the present section we will show the results obtained in [LMR09a] (Sect. 7.2.6, page 165) and [LMR09b] regarding reasoning about *CRWL* in the Isabelle/HOL proof assistant [WP06]. Fully formalizing the (meta)theory of a programming language can be beneficial for developing its foundations. There is an increasing number of researchers (see e.g. [ABF+05]) sharing the conviction that the combination *formalization+mechanized theorem proving* must (and will) play a prominent role in programming languages research and technology. In particular, formalizations help to clarify overlooked aspects, to discover pitfalls, and even to provide new insights; moreover, formalized metatheories lead to mechanized reasoning about programs, giving reliable support to tools like certifying compilers or certified program transformations.

We have chosen Isabelle/HOL as concrete logical framework for our formalization. Using such a broadly used system is not only easier, but also more flexible and stable than developing language specific tools like has been done, e.g., for logic programming [Stä98] or functional programming [dMvEP01].

In [LMR09a, LMR09b], besides formalizing the syntactic elements and the proof calculus of *CRWL* (including the support for extra variables in program rules), some basic properties of *CRWL* have been formalized and proved: closedness under c-substitutions, polarity and compositionality. Through all the proofs the Isar [WP06] scripting language has been used, developing very high level and readable proofs, pretty close to the original

"on paper" proofs.

Basic syntax of *CRWL* in Isabelle

The first step to formalize *CRWL* in Isabelle is to define elementary types for the syntactic elements.

```
datatype signat = fs string | cs string
datatype varId = vi string
datatype exp = perp | Var varId | Ap signat "exp list"
types
  subst = "varId ⇒ exp option"
  rule = "exp * exp"
  program = "rule set"
```

Signatures are represented by a datatype that provides two constructors `cs` and `fs` to distinguish between constructor and function symbols. The type `varId` is used to represent variable identifiers, which will be employed to define substitutions. Then the datatype `exp` is naturally defined following the inductive scheme of $Exp_\perp$, therefore with this representation every expression is partial by default.

Substitutions (type `subst`) are represented as partial functions from variable identifiers to expressions, using Isabelle's `option` type. Hence the domain of a substitution ϑ will be the set of elements from `varId` for which ϑ returns some value different from `None`. Note that this representation does not ensure that domains of substitutions are finite. Our proofs do not rely on this finiteness assumption. We define a function `apSubst :: "subst ⇒ exp ⇒ exp"` for applying a substitution to an expression. The composition of substitutions is defined through a function `substComp :: "subst ⇒ subst ⇒ subst"`.

Finally we represent a program rule as a pair of expressions, where the first element is considered the left-hand side of the rule and the second the right-hand side, and a program simply as a set of program rules. The set of valid *CRWL* programs is characterized by a predicate `crwlProgram :: "program ⇒ bool"` that checks whether the restrictions of left-linearity and constructor discipline are satisfied.

Just as ML, the Isabelle type system does not support subtyping, which could have been useful to represent the sets of c-terms and c-substitutions. Instead, we define predicates `cterm` and `csubst` characterizing these subtypes, which will be used as an additional hypothesis for those lemmas where some elements of the subtype are used.

Approximation order and contexts

Two key notions of *CRWL* have not yet been formalized: the approximation order $\sqsubseteq$, which will be used in the formulation of the polarity of *CRWL*, and the notion of one-hole context, which will be used in the compositionality.

The following inductively defined predicate `ordap` (with concrete infix syntax $\sqsubseteq$) models the approximation order.

```
inductive
  ordap :: "exp ⇒ exp ⇒ bool" ("_ ⊑ _" [51,51] 50)
where
  B: "perp ⊑ e"
| V: "Var x ⊑ Var x"
```

```
| Ap: "⟦ size es = size es' ; ALL i < size es. es!i ⊑ es'!i ⟧
      ⟹ Ap h es ⊑ Ap h es'"
```

Rule `B` asserts that `perp` $\sqsubseteq$ `e` holds for every `e`; rule `V` is needed for $\sqsubseteq$ to be reflexive; finally rule `Ap` ensures closedness under Σ-operations, and thus compatibility with context [BN98], because $\sqsubseteq$ is reflexive and transitive, as we will see. The following results state that our formulation of $\sqsubseteq$ defines a partial order.

```
definition ordap_less ("_ ⊏ _" [51,51] 50) where
  "e ⊏ e' ≡ e ⊑ e' ∧ e ≠ e'"
interpretation exp : order [ordap ordap_less]
```

Isabelle is able to proof that `ordap` can be interpreted as a partial order (`order` in Isabelle terminology) by using the lemmas proving the reflexivity, transitivity and anti-symmetry of `ordap` whose formulation and proof can be found in [LMR09a, LMR09b]. Contexts are represented as the datatype `cntxt`, defined as follows:

```
datatype cntxt = Hole | Cperp | CVar varId
               | CAp signat "cntxt list"
```

Note that `cntxt` cannot follow the inductive structure of *Cntxt* with precision, because the type system of Isabelle is not expressive enough to allow us to specify that only one of the arguments of `CAp` will be a context and the others will be expressions. Then our contexts are defined as expressions with possibly some holes inside. Therefore the datatype `cntxt` represents contexts with any number of holes, even zero holes, and the function `apCon :: "exp ⇒ cntxt ⇒ exp"` is defined so it puts the argument expression in every hole of the argument context. In order to characterize contexts with just one hole, we define a function `numHoles :: "cntxt ⇒ nat"` that returns the numbers of holes in a context.

The *CRWL* logic in Isabelle/HOL

The *CRWL* logic has been formalized through the inductive predicate `clto` with infix notation `"_ ⊢ _ → _"`. The rules defining `clto` faithfully follow the inductive structure of the definition of *CRWL*.

```
inductive
  clto :: "program ⇒ exp ⇒ exp ⇒ bool"  ("_ ⊢ _ → _" [100,50,50] 38)
where
  B[intro]:  "prog ⊢ exp → perp"
| RR[intro]: "prog ⊢ Var v → Var v"
| DC[intro]: "⟦size es = size ts;
               ∀i < size es. prog ⊢ es!i → ts!i
              ⟧ ⟹ prog ⊢ Ap (cs c) es → Ap (cs c) ts"
| OR[intro]: "⟦(Ap (fs f) ps, r) ∈ prog ; csubst ϑ ;
               size es = size ps ;
               ∀i < size es. prog ⊢ es!i → apSubst ϑ (ps!i);
               prog ⊢ apSubst ϑ r → t
              ⟧ ⟹ prog ⊢ Ap (fs f) es → t"
```

Using `clto` we can easily define the *CRWL* denotations in Isabelle as follows.

```
definition den :: "program ⇒ exp ⇒ exp set" where
  "den P e = {t. P ⊢ e → t}"
```

Reasoning about *CRWL* in Isabelle

The following are our formulations in Isabelle for the previously announced results: closedness under c-substitutions, polarity and compositionality. We refer the reader to [LMR09a] (Sect. 7.2.6, page 165) and [LMR09b] for details about the Isar proofs.

```
theorem crwlClosedCSubst :
  assumes "prog ⊢ e → t" and "csubst ϑ"
  shows "prog ⊢ apSubst ϑ e → apSubst ϑ t"

theorem crwlPolarity :
  assumes "prog ⊢ e → t" and "e ⊑ e'" and "t' ⊑ t"
  shows "prog ⊢ e' → t'"

theorem compCRWL :
   assumes "oneHole xC"
   shows "den P (apCon e xC) = (⋃t∈den P e. den P (apCon t xC))"
```

Reasoning about *CRWL* in Isabelle/Isar/HOL: conclusions

In this section we have reported our first experiences with the use of the Isabelle proof assistant for reasoning about *CRWL*, and presented the formalization in Isabelle/Isar/HOL of the essentials of *CRWL* obtained during these experiments. We have chosen Isabelle for its stability and its extensive libraries. Furthermore the Isar proof language allowed us to structure the proofs so that they become quite elegant and readable, as can be observed by looking at the Isabelle code.

Our formalization is generic with respect to syntax, in the sense that a previously given signature and program is not assumed, and includes important auxiliary notions like substitutions or contexts. This is in contrast to previous work [CLLF04, CP06] that focused on formalizing the semantics of each concrete program, as a way of proving concrete program properties. In contrast, our paper focuses on developing the metatheory of the formalism, allowing us to obtain results that are more general and also more powerful: we formally prove essential properties of the paradigm like *polarity* or *compositionality* of the *CRWL*-semantics. Of course, such general properties hold for each concrete program, but nothing similar was achieved in the above mentioned previous works. Hence a possible line of future work could be extending our theories so that we will be able to reason about properties of concrete programs in the line of [CLLF04, CP06].

While developing the formalization we realized an interesting fact not pointed out before: properties like polarity or compositionality do not depend on the constructor discipline and left-linearity imposed to programs, as there is no need to use the predicate `crwlProgram` as an hypothesis for those results. However, such requirements should certainly play an essential role when extending this work to formalize the adequacy of *let*-rewriting wrt. *CRWL*, one of our intended subjects of future work. We think this could be interesting in several ways. First of all it would be a further step in the direction of challenge 3 of [ABF+05], "Testing and Animating wrt. the Semantics", because we would end up getting an interpreter of *CRWL* during the process. We should then also formalize the evaluation strategy for the operational semantics, obtaining an Isabelle proof of its optimality. Finally there are precedents—see sections 3.2.4 (page 56) and 3.3.2 (page 74)—of

how the combination of a denotational and operational perspective is useful for general semantic reasoning in FLP.

3.3 Run-time Choice

"Ask me, then, if I believe in the spirit of the things as they were used, and I'll say yes. They're all here. All the things which had uses. All the mountains which had names. And we'll never be able to use them without feeling uncomfortable. And somehow the mountains will never sound right to us; we'll give them new names, but the old names are there, somewhere in time, and the mountains were shaped and seen under those names. The names we'll give to the canals and mountains and cities will fall like so much water on the back of a mallard. No matter how we touch Mars, we'll never touch it. And then we'll get mad at it, and you know what we'll do? We'll rip it up, rip the skin off, and change it to fit ourselves."

Ray Bradbury — The Martian Chronicles - 1950

3.3.1 A Fully Abstract Semantics for Constructor Systems

Here we will show our proposal for a new rewriting logic for the classical notion of term rewriting—which was presented in Sect. 3.1.1 (page 19)—, that defines a semantics that is fully abstract wrt. natural observational notions that use term rewriting as their operational notion, as it was first presented in [LRS09b] (Sect. 7.1.5, page 139).

We have already seen that CS's are useful for modelling programing languages, but they can also be used for system representation in general. These models often are value-based in the sense that we are interested in computing the values of expressions which may represent, for example, the result of an arithmetic operation, or an arrangement of the eight queens, the secret in an intruder analysis of a security protocol, the empty clause in a first-order theorem prover ... The notion of value could be made concrete in different manners: constructor terms, outer constructor part of expressions or normal forms. So in the case of CS's, an 'obvious' notion of semantics comes from defining the denotation of an expression e as the set of values reachable from e by rewriting. Two questions arise in relation to any semantics:

- Is the semantics compositional? In our case: is the semantics of an expression determined by the semantics of its subexpressions?
- Does it capture observational equivalence? That is: for two semantically equivalent expressions e, e', is it ensured that we will *observe* the same behavior when e, e' are put in the same context? This depends on a criterion of what can be observed from an expression. In the constructor discipline point of view, one is mostly interested again in observing which constructor terms (or outer stable constructor part) can be reached by rewriting.

Somehow surprisingly, the answer to both questions is negative for the 'obvious' semantics:

Example 3.3.1 ([LRS09b] Ex. 1)**.** Consider the constructors $a/0$,$b/0$,$c/1$,$d/2$ and the non-confluent program

$$f(c(X)) \to d(X,X) \quad choice(X,Y) \to X \quad choice(X,Y) \to Y$$

The expressions $e \equiv c(choice(a,b))$ and $e' \equiv choice(c(a),c(b))$ reach by rewriting exactly the same constructor values, namely $c(a)$ and $c(b)$. However, this does not ensure that e, e' behave the same when put in the same context. For instance, $f(e)$ can be rewritten to the constructor values $d(a,a), d(a,b), d(b,a), d(b,b)$ while $f(e')$ only to $d(a,a)$ and $d(b,b)$. More in general, this works starts by remarking that *knowing the constructor values of an expression e is not enough information to know the constructor values of $\mathcal{C}[e]$ for any given context $\mathcal{C}$*. The same example shows that the remark remains true if we replace 'constructor value' by 'normal form' or 'outer constructor part'. Using the terminology about full abstraction introduced in Def. 3.2.4 (page 60) of Sect. 3.2.5 all those semantics are not compositional, sound nor fully abstract.

The aim of our work can be made clear now: to define a semantics for CS's that is fully abstract (compositionality and soundness will come along the way) wrt. the observability criterion of reachable constructor terms. As it is usual when working with classical term rewriting [BN98, TeR03], we will only consider CS's without extra variables.

Our starting insight is that, to recover compositionality, the semantics must not collect a *flat* set of reachable values, like is $\{c(a), c(b)\}$ for $c(choice(a,b))$, but rather a more structured and 'packaged' representation, where constructors can be applied to sets, as to reflect more appropriately the matching capabilities of expressions. In our example, and disregarding for the moment some technical details, the denotation of $c(choice(a,b))$ will be the singleton 'package' $\{c(\{a,b\})\}$, reflecting the fact that $c(choice(a,b))$ can match $c(X)$ without reducing $choice(a,b)$, while the denotation of $choice(c(a),c(b))$ will be the two-element package $\{c(\{a\}), c(\{b\})\}$.

A semantics for CS's

Our proposed semantics has a logic flavor as it is based on a proof calculus. The use of proof calculi to specify the semantics of rewriting formalisms is not infrequent. Two well-known cases correspond to the frameworks of rewriting logic [MM02] and *CRWL* [GHLR99]. We have been inspired by the philosophy of the latter, and so the elements of our semantic domain will be finite elements, which represent partial approximations to infinite values. As it happened with *CRWL*, working with finite approximations makes it unnecessary to use a background of cpo's and powerdomains, and avoids the technical limitations of the approaches based on semantic domains with infinite (limit) elements and using fixpoint techniques [Bou81, Nys96].

Just like *CRWL* our calculus will prove statements of the form $e \twoheadrightarrow st$ expressing that st is a finite approximation to one of the possible values for e. But one fundamental difference with *CRWL* is that these will not only be *vertical approximations*, i.e., in the depth of the expressions understood as the number of outer constructors (depth in the tree that represents an expression) but also *horizontal approximations*, i.e., in the number of non-deterministic alternatives. In the following example we will try to clarify this intuitions.

Example 3.3.2. Consider the program $\mathcal{P} = \{repeat(X) \to X : repeat(X), X \;?\; Y \to X, X \;?\; Y \to Y\}$ and the constructors $: /2$ for lists and $a/0$, $b/0$ under term rewriting. Some possible vertical approximations—in *CRWL* style—to values for the expression $repeat(a)$ are $\bot$, $\bot : \bot$ and $a \;\; : \bot$. These approximations have deepened in different levels into the structure of the list and its elements. On the other hand for the expression $a \;?\; b$ we can have the values—informally using the 'packaged' representation above—$\{a\}$, $\{b\}$

and $\{a, b\}$ as possible horizontal approximations. But if we combine both ideas we can get the approximation $\{\{a\} : \{a, b\} : \{a\} : \emptyset\}$ for the expression $repeat(a\ ?\ b)$, which approximates both in the horizontal and vertical dimensions. This will be the kind of finite approximations that we will use in our semantics.

The idea of nesting of sets inside constructor symbols is not new, and similar concepts have been used in [Wad85, HO90, BEØ93, AIM02, BH07, LRS07c]. The point of this work is the application of this idea to the definition of a semantics for CS's based on finite approximations that not only is adequate for term rewriting but it is also compositional and fully abstract wrt. natural observation notions based on term rewriting. Our calculus has been designed to have a 'compositional' aspect, which makes the proof calculus itself a great aid to prove the compositionality of the semantics. The interested reader may find in [LRS09b] (Sect. 7.1.5, page 139) a more detailed discussion about related work.

SCTerms: the pieces of the semantics The 'packaged' representation we introduced intuitively above is concreted in the notion of SCTerm, which constitute the domain of our semantics.

$$\begin{aligned} &ESCTerm \ni est ::= X \mid c(st_1, \ldots, st_n) \\ &\qquad \text{for } X \in \mathcal{V}, c \in CS^n, st_1, \ldots, st_n \in SCTerm \\ &SCTerm \ni st ::= \emptyset \mid \{est_1, \ldots, est_n\} \\ &\qquad \text{for } n > 0, est_1, \ldots, est_n \in ESCTerm \end{aligned}$$

A *s-cterm* is a finite set of *elemental s-cterms*, which is an extension of the notion of c-term but having sets of values as the arguments of symbols, instead of just single values. As a set of values is precisely a s-cterm, then s-cterms and elemental s-cterms are defined by mutual recursion. A s-cterm represents an alternative between the different possibilities that are the elemental s-cterms that it is composed, while an elemental s-cterms is singleton at its root but may have several alternatives inside its arguments.

We can easily extend these notions to s-expressions by allowing the use of function symbols: $ESExp \ni ese ::= X \mid h(se_1, \ldots, se_n)$; $SExp \ni se ::= \emptyset \mid \{ese_1, \ldots, ese_n\}$, for $X \in \mathcal{V}$, $h \in \Sigma^n$, $se_1, \ldots, se_n \in SExp$, $n > 0$, $ese_1, \ldots, ese_n \in ESExp$. Note that in this context the semantic value $\emptyset$ plays the same role as $\bot$ in the *CRWL* framework, therefore s-cterms and s-expressions should be understood as *partial*.

The set $SSubst$ of *s-substitutions* consists of mappings $\sigma : \mathcal{V} \to SExp$ having a finite domain, where $dom(\sigma) = \{X \mid \sigma(X) \neq \{X\}\}$. Notice that s-substitutions replace variables by s-expressions (which are sets), and some care must be taken when extending s-substitutions to $ESExp$ and $SExp$:

$$\begin{aligned} &\sigma : ESExp \to SExp \\ &X\sigma = \sigma(X) \\ &h(\overline{se})\sigma = \{h(\overline{se\sigma})\} \end{aligned} \qquad\qquad \begin{aligned} &\sigma : SExp \to SExp \\ &\{ese_1, \ldots, ese_n\}\sigma = \textstyle\bigcup_{i \in \{1..n\}} ese_i\sigma \end{aligned}$$

The set $SCSubst$ of *s-csubstitutions* consists of mappings $\sigma : \mathcal{V} \to SCTerm$ with a finite domain, that extend to $ESCTerm$ and $SCTerm$ analogously to the case of s-substitutions. One hole (elemental) *s-contexts* are defined as:

$$sCntxt \ni s\mathcal{C} ::= [\,] \mid \{\ldots, h(\ldots, s\mathcal{C}, \ldots), \ldots\} \quad \text{with } h \in \Sigma \text{ and } s\mathcal{C} \in sCntxt$$

The application of a context to a s-expression is defined in the natural way. Notice that s-contexts allow the hole to be only in the place of a sub-s-expression. For example, the possible s-contexts of $\{Y, c(\{X\})\}$ are $[\,]$ and $\{Y, c([\,])\}$, but not $\{[], c(\{X\})\}$ nor $\{Y, []\}$.

$$\textbf{E} \quad se \twoheadrightarrow \emptyset \qquad\qquad \textbf{RR} \quad \{X\} \twoheadrightarrow \{X\} \quad \text{if } X \in \mathcal{V}$$

$$\textbf{DC} \quad \frac{se_1 \twoheadrightarrow st_1 \ \ldots \ se_n \twoheadrightarrow st_n}{\{c(se_1,\ldots,se_n)\} \twoheadrightarrow \{c(st_1,\ldots,st_n)\}} \quad \text{if } c \in CS$$

$$\textbf{More} \quad \frac{se \twoheadrightarrow st_1 \ \ldots se \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \ldots \cup st_n}$$

$$\textbf{Less} \quad \frac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots \ \{esa_m\} \twoheadrightarrow st_m}{\{ese_1,\ldots,ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m} \quad \begin{array}{l}\text{if } n \geq 2, m > 0, \text{ for any} \\ \{esa_1,\ldots,esa_m\} \subseteq \{ese_1,\ldots,ese_n\}\end{array}$$

$$\textbf{ROR} \quad \frac{se_1 \twoheadrightarrow \tilde{p_1}\theta \ \ldots \ se_n \twoheadrightarrow \tilde{p_n}\theta \ \ \tilde{r}\theta \twoheadrightarrow st}{\{f(se_1,\ldots,se_n)\} \twoheadrightarrow st} \quad \begin{array}{l}\text{if } (f(p_1,\ldots,p_n) \to r) \in \mathcal{P} \\ \theta \in SCSubst\end{array}$$

Figure 3.17: A proof calculus for constructor systems [LRS09b]

The preorder $\sqsubseteq$ is defined for s-expressions as the least preorder satisfying: $se \sqsubseteq se'$ if $\forall ese \in se.\exists ese' \in se'$ such that $ese \sqsubseteq ese'$, where for elemental s-expressions $\sqsubseteq$ is defined as the least preorder such that: $X \sqsubseteq X$ for any $X \in \mathcal{V}$ and $h(se_1,\ldots,se_n) \sqsubseteq h(se'_1,\ldots,se'_n)$ iff $se_i \sqsubseteq se'_i$ for $i = 1..n$. For s-substitutions, the preorder is defined as $\sigma \sqsubseteq \sigma'$ iff $\forall X \in \mathcal{V}, \sigma(X) \sqsubseteq \sigma'(X)$. Besides, the preorder $\trianglelefteq$ over $SSubst$, is defined as $\sigma \trianglelefteq \sigma'$ iff $\forall X \in \mathcal{V}, [\![\sigma(X)]\!] \subseteq [\![\sigma'(X)]\!]$.

The proof calculus of the next section needs to use function rules transformed into the new syntactical framework of s-expressions. For this purpose we define the transformation of $e \in Exp$ into a s-expression $\tilde{e} \in SExp$ as: $\tilde{\bot} = \emptyset$; $\tilde{X} = \{X\}$ for any $X \in \mathcal{V}$; $\widetilde{h(e_1,\ldots,e_n)} = \{h(\tilde{e_1},\ldots,\tilde{e_n})\}$, with $h \in \Sigma^n$. The transformation $\tilde{\mathcal{C}}$ of a context $\mathcal{C}$ is defined in the natural way, so that it verifies $\widetilde{\mathcal{C}[e]} = \tilde{\mathcal{C}}[\tilde{e}]$. On the other hand, $\tilde{\sigma}$ is defined as $\tilde{\sigma}(X) = \widetilde{\sigma(X)}$, for $\sigma \in Subst$.

A Proof Calculus As we pointed out before, our proof calculus follows the style of *CRWL*, so it proves reduction statements of the form $se \twoheadrightarrow st$ with $se \in SExp$ and $st \in SCTerm$, expressing that st represents a finite approximation to one of the possible structured sets of values for se. We cannot just prove statements of the form $e \twoheadrightarrow st$ with $e \in Exp$ because of the intensive use of $SCSubst$ in the calculus, which may introduce sets in the expressions during parameter passing, even when starting the computation from an ordinary $e \in Exp$.

Just as *CRWL*, to achieve a compositional aspect of the calculus expressions are evaluated in an innermost way, and the use of any transitivity rule is avoided. By the use of partial s-cterms as values, the 'compositional' innermost procedure of the calculus does not enforce strictness of functions, which is essential to achieve completeness of our semantics wrt. term rewriting even for non-terminating CS's. The calculus is presented in Fig. 3.17. Rules E (empty), RR and DC mimic the corresponding rules of *CRWL* (B in the case of E). The novelties are: More, which allows us to compute more than one value for a s-expression, and to collect these values together; Less, which allows us to discard some elemental s-expressions from the s-expression under evaluation; and finally rule ROR (run-time outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of a $SCSubst$ θ) and then reduce the instantiated right-hand side. The use of $SCSubst$s is fundamental to get the exact behaviour of term rewriting, because then the branching

information associated to the computation of each $\tilde{p_i}\theta$ is not lost in some kind of flattening to a set of c-terms, but kept into the structured representation of $SCTerm$s.

We write $\mathcal{P} \vdash se \twoheadrightarrow st$ to express that $se \twoheadrightarrow st$ is derivable in our calculus under the CS $\mathcal{P}$. The *denotation* of a s-expression se under $\mathcal{P}$ is defined as $[\![se]\!]^{\mathcal{P}} = \{st \in SCTerm \mid \mathcal{P} \vdash se \twoheadrightarrow st\}$. In the following we will usually omit the reference to $\mathcal{P}$.

Example 3.3.3 ([LRS09b] Ex. 2)**.** Consider the CS of Ex. 3.3.1. We can use our calculus to prove the statement $f(c(\widetilde{choice}(a,b))) \twoheadrightarrow \widetilde{d(a,b)}$ (some steps have been omitted for the sake of conciseness, and *choice* is abbreviated to *ch*):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{a\} \twoheadrightarrow \{a\}}\,\text{DC}\quad\dfrac{}{\{b\} \twoheadrightarrow \emptyset}\,\text{E}\quad\dfrac{\cdots}{\{a\} \twoheadrightarrow \{a\}}}{\{ch(\{a\},\{b\})\} \twoheadrightarrow \{a\}}\,\text{ROR}\quad\dfrac{\cdots}{\{ch(\{a\},\{b\})\} \twoheadrightarrow \{b\}}\,\text{ROR}}{\{ch(\{a\},\{b\})\} \twoheadrightarrow \{a,b\}}\,\text{More}}{\{c(\{ch(\{a\},\{b\})\})\} \twoheadrightarrow \{c(\{a,b\})\}}\,\text{DC}\quad\dfrac{(*)}{\{d(\{a,b\},\{a,b\})\} \twoheadrightarrow \{d(\{a\},\{b\})\}}\,\text{DC}}{f(c(\widetilde{ch(a,b)})) \equiv \{f(\{c(\{ch(\{a\},\{b\})\})\})\} \twoheadrightarrow \{d(\{a\},\{b\})\} \equiv \widetilde{d(a,b)}}\,\text{ROR}$$

where $(*)$ is the derivation:

$$\dfrac{\dfrac{\dfrac{}{\{a\} \twoheadrightarrow \{a\}}\,\text{DC}}{\{a,b\} \twoheadrightarrow \{a\}}\,\text{Less}\quad\dfrac{\cdots}{\{a,b\} \twoheadrightarrow \{b\}}}{\{d(\{a,b\},\{a,b\})\} \twoheadrightarrow \{d(\{a\},\{b\})\}}\,\text{DC}$$

On the other hand, $\widetilde{d(a,b)}$ is not a correct value for $f(\widetilde{choice(c(a),c(b))})$, because in that expression the evaluation of $\widetilde{choice(c(a),c(b))}$ has to be performed in order to get a value matching the argument of the left-hand side of the only rule for f, and the only matching values for it are $\widetilde{c(a)}$, $\widetilde{c(b)}$ and $\{c(\emptyset)\}$, as for example $\{c(\{a\}), c(\{b\})\}$ does not match $\widetilde{c(X)}$.

Our calculus enjoys the following nice properties:

Proposition 3.3.1 (Polarity, [LRS09b] Prop. 1)**.** Let $se, se' \in SExp$, $st, st' \in SCTerm$. If $se \sqsubseteq se'$ and $st' \sqsubseteq st$ then $st \in [\![se]\!]$ implies $st' \in [\![se']\!]$.

Proposition 3.3.2 (Monotonicity of substitutions, [LRS09b] Prop. 2)**.** Let $se \in SExp$, $\sigma, \sigma' \in SSubst$. If $\sigma \trianglelefteq \sigma'$ or $\sigma \sqsubseteq \sigma'$ then $[\![se\sigma]\!] \subseteq [\![se\sigma']\!]$.

Theorem 3.3.1 (Compositionality, [LRS09b] Th. 1)**.** *For all $s\mathcal{C} \in sCntxt, se \in SExp$,*

$$[\![s\mathcal{C}[se]]\!] = \bigcup_{st \in [\![se]\!]} [\![s\mathcal{C}[st]]\!]$$

As a consequence: $[\![se]\!] = [\![se']\!] \Leftrightarrow \forall s\mathcal{C}.\ [\![s\mathcal{C}[se]]\!] = [\![s\mathcal{C}[se']]\!]$.

Regarding *closedness* under substitutions, as we use $SCSubst$ for parameter passing it is natural to have closedness of reductions under this type of substitutions. Besides, as rewriting is closed under $Subst$, it is expected to have some kind of closedness for $Subst$ too. But in general it is not true that for any $st \in SCTerm, \sigma \in SSubst$ we have $st\sigma \in SCTerm$, therefore it makes no sense to expect that $se \twoheadrightarrow st$ implies $se\sigma \twoheadrightarrow st\sigma$, as the reductions in our logic are from $SExp$ to $SCTerm$. Nevertheless we still can say something about that, as we can see in the following property.

Proposition 3.3.3 (Closedness under substitutions, [LRS09b] Prop. 3)**.** Let $se \in SExp$ and $st \in [\![se]\!]$. Then: *a)* $\forall \theta \in SCSubst, st\theta \in [\![se\theta]\!]$. *b)* $\forall \sigma \in SSubst, [\![st\sigma]\!] \subseteq [\![se\sigma]\!]$.

All these properties are powerful tools to reason about the denotation of s-expressions. And this reasoning power is transferred to the term rewriting universe through the adequacy results that we will see in the next section, thus opening paths for the development of new reasoning techniques for CS's.

Relation with rewriting

The nice properties of our logic would be useless unless they are accompanied by strong adequacy results with respect to the term rewriting relation. We first address the *completeness* of our logic, i.e., that the semantics of any expression captures any c-term reachable from it by rewriting. As a first result we have:

Proposition 3.3.4 ([LRS09b] Prop. 4)**.** For all $e, e' \in Exp$, if $e \to^* e'$ then $[\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$.

The keys for its proof are Th. 3.3.1 and the following Lemma 3.3.1, expressing that any reduction $se\sigma \twoheadrightarrow st$ needs to use only a finite amount of the information contained in σ, formalized through the notion of denotation of a $SSubst$, defined as $[\![\sigma]\!] = \{\theta \in SCSubst \mid \forall X \in \mathcal{V}, \sigma(X) \twoheadrightarrow \theta(X)\}$.

Lemma 3.3.1 ([LRS09b] Lemma 1)**.** *Let $\sigma \in SSubst, se \in SExp, st \in SCTerm$. If $se\sigma \twoheadrightarrow st$ then there exists $\theta \in [\![\sigma]\!]$ such that $se\theta \twoheadrightarrow st$.*

Now we can apply Prop. 3.3.4 to get the following strong completeness result.

Theorem 3.3.2 (Completeness, [LRS09b] Th. 2)**.** *For all $e, e' \in Exp, t \in CTerm$,*
a) $e \to^* e'$ implies $\widetilde{|e'|} \in [\![\tilde{e}]\!]$ b) $e \to^* t$ implies $\widetilde{t} \in [\![\tilde{e}]\!]$

We also want our logic to be *correct*, in the sense that the semantics of any expression does not compute more c-terms than those reachable by rewriting. One key ingredient will be the *domination relation* $\_ \lessdot \_$ defined in Fig. 3.18 (we will omit the prefix "$\mathcal{P} \vdash$" when deducible from the context). With this relation we try to transfer to the rewriting world the finer distinction between sets of values that the structured representation of $SCTerm$ allows us to perform. This way under the CS of Ex. 3.3.1 we have $\{c(\{a, b\})\} \lessdot c(choice(a, b))$ but not $\{c(\{a, b\})\} \lessdot choice(c(a), c(b))$. The domination relation $\_ \lessdot \_$ has a strong relation with our semantics, as stated in the following result:

Lemma 3.3.2 (Domination, [LRS09b] Lemma 2)**.** *For all $e \in Exp, st \in SCTerm$: $st \in [\![\tilde{e}]\!]$ iff $st \lessdot e$.*

Notice that $\_ \lessdot \_$ only talks about reductions for $\tilde{e}$ with $e \in Exp$, and so it cannot be used to formulate properties like those seen in Sect. 3.3.1, although it inherits them through Lemma 3.3.2. But the good thing about $\_ \lessdot \_$ is that it already has a strong connection with rewriting, as it is defined by means of rewriting derivations. Hence we can perform a simple induction on the structure of $SCTerm$ and $ESCTerm$ to prove the following result, which uses the notion of flattening defined in Sect. 3.3.1.

Lemma 3.3.3 ([LRS09b] Lemma 3)**.** *Let $st \in SCTerm$, $est \in ESCTerm$, $e \in Exp$, and assume $t \in flat(st)$. If $st \lessdot e$ then $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.*

And now we are ready to state and prove our main correctness result.

$\_ \vdash \_ \lessdot \_ \subseteq CS \times SCTerm \times Exp$	$\_ \vdash \_ \lessdot \_ \subseteq CS \times ESCTerm \times Exp$
$\mathcal{P} \vdash st \lessdot e$ if $\forall est \in st, \mathcal{P} \vdash est \lessdot e$	$\mathcal{P} \vdash X \lessdot e$ if $\mathcal{P} \vdash e \to^* X$
	$\mathcal{P} \vdash c(\overline{st}) \lessdot e$ if $\mathcal{P} \vdash e \to^* c(\overline{e})$ for some $\overline{e}$ such that $\forall e_i \in \overline{e}, \mathcal{P} \vdash st_i \lessdot e_i$

Figure 3.18: Domination relation [LRS09b]

Theorem 3.3.3 (Correctness, [LRS09b] Th. 3)**.** *Let $e \in Exp, st \in SCTerm, t \in CTerm_\perp$:*

a) If $st \in [\![\tilde{e}]\!]$ and $t \in flat(st)$, then $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.

b) If $\tilde{t} \in [\![\tilde{e}]\!]$, then $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.

c) Besides, in a) or b), if $t \in CTerm$, then $e \to^* t$.

Correctness is the point where left linearity of programs is essential. Without left linearity, the results of Sect. 3.3.1 still hold, but the semantics becomes incorrect wrt. rewriting. For instance, if $\mathcal{P} \equiv \{f(X,X) \to a\}$ and $e \equiv f(a,b)$ then it can be shown that $\tilde{a} \in [\![\tilde{e}]\!]$ but $e \not\to^* a$, contradicting Th. 3.3.3.

Full abstraction

Our semantics $[\![se]\!]^{\mathcal{P}}$ is defined for s-expressions, but induces naturally a notion of semantics for ordinary expressions $e \in Exp$:

$$[\![e]\!]^{\mathcal{P}}_S = [\![\tilde{e}]\!]^{\mathcal{P}}(= \{st \in SCTerm | \tilde{e} \twoheadrightarrow st\})$$

Here we discuss full abstraction in the context of CS's and show that $[\![\_]\!]_S$ achieves it, in contrast to semantics directly based on sets of results, informally described above.

In the next definition we collect some notions of semantics and observables for the case of CS's. Our new contributed semantics is $[\![\_]\!]_S$, the rest are the 'obvious' semantics we talked about in the introduction. As usual, we omit the program $\mathcal{P}$ in notations.

Definition 3.3.1 (Semantics and observations for CS's, [LRS09b] Def. 2)**.**
We consider the following semantics for expressions $e \in Exp$:

$$[\![e]\!]_{rw} = \{e' \mid e \to^* e'\} \qquad [\![e]\!]_{nf} = \{e' \mid e \to^* e', e' \text{in normal form}\}$$
$$[\![e]\!]_t = \{t \in CTerm | e \to^* t\} \qquad [\![e]\!]_{t_\perp} = \{t \in CTerm_\perp \mid \exists e'.(e \to^* e' \wedge t \sqsubseteq |e'|)\}$$
$$[\![e]\!]_S = [\![\tilde{e}]\!]$$

and two observation functions for expressions: $\mathcal{O}_t(e) = [\![e]\!]_t$ and $\mathcal{O}_{t_\perp}(e) = [\![e]\!]_{t_\perp}$.

The next result shows that, although $\mathcal{O}_t, \mathcal{O}_{t_\perp}$ define different observations, it is irrelevant which is chosen, as far as full abstraction is concerned.

Proposition 3.3.5 ([LRS09b] Prop. 6)**.** Assume a given semantics $[\![\_]\!]$. Then:

$$[\![\_]\!] \text{ is fully abstract wrt. } \mathcal{O}_t \Leftrightarrow [\![\_]\!] \text{ is fully abstract wrt. } \mathcal{O}_{t_\perp}.$$

Now we show that the first four semantics, $[\![\_]\!]_{rw}, [\![\_]\!]_{nf}, [\![\_]\!]_t, [\![\_]\!]_{t_\perp}$ do not have good properties. We use $\mathcal{O}_t$ in the result but, according to the previous result, $\mathcal{O}_{t_\perp}$ could be used instead.

Proposition 3.3.6 ([LRS09b] Prop. 7)**.**
(a) $[\![\_]\!]_{rw}, [\![\_]\!]_{nf}, [\![\_]\!]_t, [\![\_]\!]_{t_\perp}$ are not fully abstract wrt. $\mathcal{O}_t$.
(b) Moreover, $[\![\_]\!]_{nf}, [\![\_]\!]_t, [\![\_]\!]_{t_\perp}$ are not compositional, nor sound wrt. $\mathcal{O}_t$.
(c) $[\![\_]\!]_{nf}$ ($[\![\_]\!]_t$ resp.) remains not compositional nor sound wrt $\mathcal{O}_t$ even if programs are restricted to be confluent (confluent and terminating, resp.).

Finally, we show that our semantics does not present those problems.

Theorem 3.3.4 (Compositionality and full abstraction of $[\![\_]\!]_S$, [LRS09b] Th. 4)**.**
$[\![\_]\!]_S$ is compositional and fully abstract wrt. $\mathcal{O}_t$ and $\mathcal{O}_{t_\perp}$.

A Fully Abstract Semantics for Constructor Systems: conclusions

We have provided a semantics for constructor systems that is compositional and even fully abstract with respect to natural notions of observation that extract the outer constructor part of outcomes as relevant information of computations. To the best of our knowledge, this is the first time that full abstraction has been achieved for this class of programs and observations. Along the way to this result we have made some contributions: after noticing that 'obvious' semantics directly based on rewrite sequences lack compositionality, our main insight has been that it can be recovered by recursively packaging sets of results below constructor symbols. That insight has been realized at the technical level by introducing s-cterms as suitable semantic values, and giving a proof calculus able to derive reachable s-cterms from a given expression. Previous to full abstraction, we have proved a bunch of good properties of the semantics: polarity, compositionality, closedness under substitutions, correctness and completeness with respect to rewriting.

We expect our semantics to be a useful tool for CS-based program manipulation. We remark that, for instance, to justify the correctness of a CS-transformation by proving preservation of reachable values could be incorrect if transformations are to be used locally. Our semantics could be a better option, as illustrated by the following simple example: consider a program $\mathcal{P}$ containing the rules $f \to c(g)$, $g \to e$, $g \to e'$, where e, e' are expressions that costly reduce to a, b respectively. An 'obvious' partial evaluation might suggest replacing f's definition by the optimized one $f \to c(a)$, $f \to c(b)$, leading to a transformed program $\mathcal{P}'$, presumably equivalent to $\mathcal{P}$. This is wrong: if $\mathcal{P}$ defines also h by the rule $h(c(X)) \to d(X, X)$, then $h(f)$ behaves different—under term rewriting—with both definitions of f. This is detected in our semantics (using e.g. the variant $[\![\_]\!]_S$ of Def. 3.3.1) because $[\![f]\!]_S^{\mathcal{P}} \ni \{c(\{a, b\})\} \notin [\![f]\!]_S^{\mathcal{P}'}$. Imagine, however, that the original program piece was $f \to g$, $g \to c(e)$, $g \to c(e')$. In this case, the 'obvious' partial evaluation of f would lead again to $f \to c(a)$, $f \to c(b)$. Is this right now? Yes, because $[\![f]\!]_S^{\mathcal{P}} = [\![f]\!]_S^{\mathcal{P}'}$, and full abstraction of $[\![\_]\!]_S$ makes the rest. A deeper investigation of these issues is planned for the future.

There are other aspects not yet accomplished that can be subject of future work. In the paper, 'compositionality' refers to expressions wrt. its subexpressions, and not to programs obtained by joining others, an interesting topic related to modularity (see e.g. [AFRV93]). We also plan to investigate full abstraction for other notions of observations, in two different senses: by giving a more active role to variables (as happens in [AFRV93, ACE+03, HL01]) taking into account that, for instance, variables can be subject of narrowing substitutions; and by replacing our 'may-convergent' or 'angelic' view of non-determinism (in which two expressions may have the same semantics even if one admits divergent reductions while the other does not) by a 'must-convergence' view where divergence plays a role. Dropping the constructor restriction is also interesting, replacing the role of constructor values by appropriate alternatives.

3.3.2 Call-time Choice vs. Run-time Choice

At the beginning of Sect. 3.2.2 we saw some technical results about the impossibility of simulating call-time choice with run-time choice and vice versa, namely Ex. 3.2.1 (page 38) and Prop. 3.2.1 (page 39). In the present section we will formally prove that run-time choice is strictly bigger than call-time choice in the sense that the set of values reachable by run-time choice evaluation is bigger than those reachable using call-time choice, while the converse does not hold in general. That is, that call-time choice is sound but not

complete wrt. run-time choice, when the set of reachable values is considered. Besides, we will show that call-time choice becomes complete for deterministic programs, a property close to confluence. Technically this has been done by using traditional term rewriting as our formulation of run-time choice — as it is usual in the field — and *let*-rewriting as our formulation of call-time choice — because it is adequate to *CRWL*, which is recognized as a standard semantics for call-time choice. The programs considered are CS's with extra variables.

Finally, we examine the relationship between traditional narrowing and *let*-narrowing. As expected from its adequacy to traditional rewriting and *let*-rewriting respectively, *let*-narrowing is sound wrt. traditional narrowing and we can only grant completeness for deterministic programs.

These results were first presented in [LRS07b] (Sect. 7.1.1, page 126) and [LRS09d] (Sect. 7.2.1, page 140), where we refer the reader for further details.

Let-rewriting versus classical rewriting

Thanks to the equivalence of *CRWL* and *let*-rewriting we can choose the most appropriate point of view for each of the two goals (soundness and completeness): we will use *let*-rewriting for proving soundness, and the proof calculus of *CRWL* for defining the property of determinism and proving that, under determinism, completeness holds.

Soundness of let-rewriting w.r.t. classical rewriting Firstly, we need a syntactic transformation from $LExp$ into Exp, removing the *let* constructions (thus losing the sharing information they provide). Given $e \in LExp$ we define its transformation into a standard expression $\widehat{e}$ as $\widehat{X} \equiv X$; $\widehat{h(e_1,\ldots,e_n)} \equiv h(\widehat{e_1},\ldots,\widehat{e_n})$; $\widehat{let\ X_p = e_1\ in\ e_2} \equiv \widehat{e_2}[X_p/\widehat{e_1}]$. Using this notion we get a first soundness result, stating that what can be done in one step of *let*-rewriting, can also be done in zero or more steps of ordinary rewriting, after erasing the sharing information by the transformation $\widehat{\ }$:

Lemma 3.3.4 ([LRS07b] Lemma 12)**.** *For all $e, e' \in LExp$ we have: $e \to_l e'$ implies $\widehat{e} \to^* \widehat{e'}$.*

Some other soundness results follow easily from the lemma above. The first one expresses that any expression (not involving *let*'s) reachable by *let*-rewriting can be also reached by ordinary rewriting. In other terms, *let*-rewriting ($\to_l^*$) is a sub-relation of rewriting ($\to^*$), when ($\to_l^*$) is restricted to ordinary expressions (not involving *let*'s).

Theorem 3.3.5 ([LRS07b] Th. 8)**.** *For any $e, e' \in LExp$, $e \to_l^* e'$ implies $\widehat{e} \to^* \widehat{e'}$. As a consequence, if $e, e' \in Exp$, then $e \to_l^* e'$ implies $e \to^* e'$.*

The next result, based on the adequacy to *CRWL* of *let*-rewriting, is a soundness theorem for *CRWL* with respect to ordinary rewriting.

Theorem 3.3.6 ([LRS07b] Th. 9)**.** *For all $e \in Exp$ and $t \in CTerm_\bot$, $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ implies $\exists e' \in Exp$ such that $e \to^* e'$ and $t \sqsubseteq |e'|$.*

Completeness of *CRWL* w.r.t. classical rewriting As commented before (see Ex. 1.1.1 (page 5) and Ex. 3.1.2 (page 23)), we cannot expect to get a completeness result of the *CRWL* framework w.r.t. classical rewriting for any program, as for example using the program $\mathcal{P} = \{coin \to 0, coin \to 1, pair(X) \to (X, X)\}$ we can reach the value $(0, 1)$ for the expression $pair(coin)$ under classical rewriting, but not under *let*-rewriting. Nevertheless we will show that completeness is granted for the class of deterministic programs —a traditional notion in the *CRWL* framework [GHLR99]— which are defined as follows:

Definition 3.3.2 (Deterministic *CRWL*-program)**.**
A *CRWL*-program $\mathcal{P}$ is *deterministic* iff the denotation $[\![e]\!]^{\mathcal{P}}$ of any expression $e \in Exp_\bot$ is a directed set. In other words, iff $\forall e \in Exp_\bot$ and $t_1, t_2 \in [\![e]\!]^{\mathcal{P}}$ there exists $t_3 \in [\![e]\!]^{\mathcal{P}}$ with $t_1 \sqsubseteq t_3$ and $t_2 \sqsubseteq t_3$.

Determinism as defined here is intuitively close to confluence, but the two notions do not coincide. Determinism does not imply confluence, as the following example shows:

Example 3.3.4 ([LRS07b] Ex. 4)**.** Consider the program $\mathcal{P} = \{f \to a, f \to loop, loop \to loop\}$, where a is a constructor. It is clear that $\mathcal{P}$ is not confluent (f can be reduced to a and $loop$, which cannot be joined to a common reduct), but is deterministic, since $[\![f]\!]^{\mathcal{P}} = \{\bot, a\}$, $[\![loop]\!]^{\mathcal{P}} = \{\bot\}$ and $[\![a]\!]^{\mathcal{P}} = \{\bot, a\}$, each of them being a directed set.

We conjecture that the reverse implication (confluence $\Rightarrow$ determinism) is true, but a precise proof of this fact seems surprisingly complicated and we have not yet completed it.

Determinism has been defined as a semantic property. However, thanks to the equivalence of *CRWL* and *let*-rewriting, it can be also characterized in terms of reduction, as the following result shows:

Lemma 3.3.5 ([LRS07b] Lemma 13)**.** *A CRWL-program $\mathcal{P}$ is deterministic iff for any expressions $e, e', e'' \in Exp$ with $e \to_l^* e'$ and $e \to_l^* e''$, there exists $e''' \in Exp$ such that $e \to_l^* e'''$ and $|e'''| \sqsupseteq |e'|, |e'''| \sqsupseteq |e''|$.*

Under the hypothesis of deterministic programs, we get the following first completeness result:

Lemma 3.3.6. *For any CRWL-program $\mathcal{P}$, if it is deterministic then for all $e, e' \in Exp$, $\mathcal{P} \vdash e \to^* e'$ implies $[\![e']\!]^{\mathcal{P}}_{CRWL} \subseteq [\![e]\!]^{\mathcal{P}}_{CRWL}$.*

The previous lemma, together with the equivalence of *CRWL* and *let*-rewriting given by Theorem 3.2.12, allows to obtain strong relationships between rewriting, let-rewriting and *CRWL*, for the class of deterministic programs.

Theorem 3.3.7.
Let $\mathcal{P}$ be a deterministic CRWL-program, $e, e' \in Exp, t \in CTerm$. Then:
a) $e \to^* e'$ implies $e \to_l^* e''$ for some $e'' \in LExp$ with $|e''| \sqsupseteq |e'|$.
b) $e \to^* t$ iff $e \to_l^* t$ iff $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$.

Notice that in part a) we cannot ensure $e \to^* e'$ implies $e \to_l^* e'$, because rewriting can reach some intermediate expressions not reachable by *let*-rewriting. For instance, given the deterministic program with the rules $g \to a$ and $f(x) \to c(x, x)$, we have $f(g) \to^* c(g, a)$, but not $f(g) \to_l^* c(g, a)$. Still, part a) is a strong completeness result for let-rewriting wrt. rewriting for deterministic programs, since it says that the outer constructed part obtained in a rewriting derivation can be also obtained or even refined in a *let*-derivation.

Combined with Theorem 3.3.5, part a) expresses a kind of equivalence between *let*-rewriting and rewriting, valid for general derivations, even non-terminating ones. For terminated derivations reaching a constructor term (not further reducible), part b) gives an even stronger equivalence result.

Let-narrowing versus classical narrowing

As announced before, joining Th. 3.3.7 with the results of Sect. 3.2.3 we can easily establish some relationships between *let*-narrowing and ordinary rewriting/narrowing, as follows (we assume here that all involved substitutions are admissible):

Theorem 3.3.8 ([LRS09d] Th. 3)**.** *Let $\mathcal{P}$ be any program, $e \in Exp, \theta \in CSubst, t \in CTerm$. Then:*

(a) *$e \leadsto_\theta^{l^*} t$ implies $e\theta \to^* t$.*

(b) *If in addition $\mathcal{P}$ is deterministic, then:*

- **(b_1)** *If $e\theta \to^* t$, there exist $t' \in CTerm$, $\sigma, \theta' \in CSubst$ such that $e \leadsto_\sigma^{l^*} t'$, $t'\theta' = t$ and $\sigma\theta' = \theta[var(e)]$ (and therefore $t' \preceq t, \sigma \preceq \theta[var(e)]$).*
- **(b_2)** *If $e \leadsto_\theta^* t$, the same conclusion of (b_1) holds.*

Part *(a)* expresses soundness of $\leadsto^l$ wrt. rewriting, and part *(b)* is a completeness result for $\leadsto^l$ wrt. rewriting/narrowing, for the class of deterministic programs.

Call-time Choice vs. Run-time Choice: conclusions

We have proved that for deterministic programs (a semantic condition very close to confluence) let-rewriting (hence *CRWL*-derivability) and ordinary rewriting coincide in a precise technical sense, while in general let-rewriting is a sub-relation of rewriting. We stress the fact that this is a new, technically non-trivial result connecting the *CRWL* and rewrite worlds; to the best of our knowledge, this kind of results were completely missing in the literature. Furthermore, we strongly conjecture (and we are hopefully very close to a complete proof) that confluence of a *CRWL* -program (in the ordinary sense of TRS's) implies semantic determinism, which will imply that under confluence rewriting and let-rewriting are equivalent in some technical sense.

We have also proved soundness of *let*-narrowing wrt. ordinary rewriting and completeness for the wide class of deterministic programs, thus giving a technical support to the intuitive fact that combining sharing with narrowing does not create new answers when compared to classical narrowing, and at the same time does not lose answers in case of deterministic systems. As far as we know these results are new in the narrowing literature.

Therefore the most important remaining question is the very intuitive (but hard to prove) result stating that confluence implies determinism of programs. We think that the semantics for CS's developed in Sect. 3.3.1 could be a crucial tool to prove this result, as it makes the connection between term rewriting and the proof calculus based semantic world of *CRWL*.

3.3.3 Combining Call-time and Run-time Choice Parameter Passing

We have already commented that the semantic option of call-time choice, as it is specified by the *CRWL* logic, is implemented by current FLP languages like *Toy* [LS99, CSe06] or *Curry* [Han06]. On the other hand, although run-time choice is the option chosen in non-deterministic specification languages like Maude [CDE+07], CafeOBJ [FN97] or Elan [vdBMR02], it has been rarely [Ant97] thought as a valuable global alternative to call-time choice for the value-based view of FLP. However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal, being a typical example the modeling of grammars for parsing, where a function to define the Kleene's $*$ operator is very convenient, and only can be to defined using a run-time choice semantics.

Another interesting example comes from the challenges regarding the implementation of type classes in FLP through the classical transformational technique of [WB89] pointed out by Lux in [Lux09]. There the use of run-time choice parameter passing for the dictionary structure containing the instance class implementation of a data is needed to avoid an unwanted extra sharing of non-deterministic constant member functions, that would cause a loss of results.

In this section we will present two different approaches for the combination of call-time choice and run-time choices developed by us in [LRS09a] (Sect. 7.1.4, page 139) and [LRS09c] (Sect. 7.2.4, page 140). The advantage of those approaches is that both call-time choice and run-time choice have clean and high level formulations —the *CRWL* logic and term rewriting, respectively— which are also consolidated frameworks, thus making programs written using that semantics combination easy to understand, at least for a reader used to declarative programming or to formal methods in general. This is also stressed by the fact that the proposals of [LRS09a, LRS09c] are conservative extensions of either pure run-time choice or pure call-time choice.

In a run-time choice environment

The proposal of [LRS09a] starts from the run-time choice framework of term rewriting an extends it with primitives to express sharing of computations, and therefore call-time choice.

User programs are CS's with extra variables but where each function in the program is annotated with the intended semantics (run-time choice or call-time choice) for it. This annotated programs are then transformed into a core language that essentially results of adding a *let*-construct to a run-time choice framework (i.e., to ordinary rewriting). For this core language we define a rewriting relation (called *rt-let*-rewriting) that mixes ordinary rewriting with suitable rules for the propagation of bindings contained in *let*s. Our language subsumes pure run-time choice and pure call-time choice (it is enough to annotate all functions with the corresponding semantics). Since those are well-established frameworks, we must ensure that our transformation into the core language is harmless and respects the original semantics.

Hence this proposal is double in the sense that it allows the combination of call-time and run-time choice at two different abstraction levels: either at the level of CS's with annotated functions, or at the level of the core language governed by *rt-let*-rewriting.

For the sake of clarity, here we rename the '*let*-rewriting' relation of Sect. 3.2.2 to '*ct-let*-rewriting', to distinguish it from *rt-let*-rewriting. As we will see later, although *ct-let*-rewriting and *rt-let*-rewriting share the same syntax—both of them manipulate *let*-expressions—, their behaviour is very different, and in particular they are incomparable step by step. Hence it is important to have this distinction in mind during this section for a propper understanding of the *rt-let*-rewriting relation.

Syntax of annotated programs The set FS of function symbols is partitioned into two sets FS_{rtc} and FS_{ctc} of functions following run-time choice and call-time choice respectively. We assume that function symbols of FS_{rtc} (FS_{ctc} resp.) are introduced by declarations of the form `rtc f` (`ctc f` resp.). Therefore a user program is a tuple where the first component is a regular CS's with extra variables, and the second and third one are the sets FS_{rtc} and FS_{ctc}, respectively. By $\mathcal{P}_{rc}$ we denote the set of all the possible tuples of that shape.

Example 3.3.5 ([LRS09a] Ex. 1)**.** Modeling grammar rules for string generation can be directly done by CS's like the following (non-confluent and non-terminating) one, in which we assume that texts (terminals) are represented as strings, identified with lists of characters, that can be concatenated with the infix operation `++` (defined in a standard way):

```
letter → "a"  ......  letter → "z"
word → " "            word → letter ++ word
```

The program acts as a non-deterministic generator of the strings in the language defined by the grammar. Each individual reduction leads to a string. Now imagine that we want to include the generation of palindromes in the specification. This could be done by the rewrite rules:

```
palindrome → palAux(word)
palAux(X) → X ++ reverse(X)
palAux(X) → X ++ letter ++ reverse(X)
```

where `reverse` is defined in any standard way. It is important to remark that the definition of `palindrome/palAux` works fine only if call-time choice is adopted for non-determinism, meaning operationally that in the (partial) reduction

```
palindrome → palAux(word) → word ++ reverse(word)
```

the two occurrences of `word` created by the rule of `palAux` must be shared. If run-time choice (i.e., ordinary rewriting) were used, the two occurrences of `word` could follow independent ways, and therefore `palindrome` could be reduced, for instance, to `"oops"`, which is not a palindrome.

Two useful operators to structure grammar specifications are the alternative '|' and Kleene's '$*$' for repetitions:

```
X | Y → X       X | Y → Y
star(X) → " "   star(X) → X ++ star(X)
```

With them `letter` and `word` could be redefined as follows:

```
letter → "a" | "b" | ...  | "z"
word → star(letter)
```

The annoying fact is that this does not work! At least not under call-time choice, with which all the occurrences of `letter` created by `star` will be shared and therefore `word` will only generate words like *aaa* or *nnnn*, made with repetitions of the same letter. The problem would be overcome if the function `star` would follow a *run-time choice* regime, so that the evaluation of each of the two occurrences of `letter` created in the rewrite sequence

$$\texttt{word} \rightarrow \texttt{star(letter)} \rightarrow \texttt{letter ++ star(letter)}$$

is not shared, but could evolve independently.

We conclude that in this example neither call-time nor run-time choice are a good single option as semantics for the whole program. The definition of `palAux` requires call-time choice, while `star` requires run-time choice. Hence a suitable partition of the set of function symbols would have $FS_{rtc} = \{star, |\}$, since they are operations intended to be applied to generators of strings (although in the case of |, the behavior would not change if declared as `ctc`). FS_{ctc} would consist of the remaining functions in the program. Therefore our program is $(\mathcal{P}, \{star, |\}, \{letter, word, palindrome, palAux, reverse\})$ where $\mathcal{P}$ is the CS consisting of the rules above.

Semantics of annotated programs: the core language Our next step is defining the expected behavior of programs. Since programs are annotated CS's, this is best done by giving a precise notion of reduction step that generalizes adequately the standard notion of rewrite step. If `rtc`-functions and `ctc`-functions did not interact, their combination would not be a challenge at all: `rtc`-function applications would be reduced using ordinary rewriting, and for `ctc`-functions we could use any of the existing formal descriptions of call-time choice [GHLR99, AHH+05, LRS07b]. However, this is not enough if computations merge both kinds of functions; this happens easily, as we have just seen in Ex. 3.3.5 where the evaluation of `palindrome` (a `ctc`-function) involves the evaluation of `star` (a `rtc`-function) though the evaluation of `word`.

We have found to be convenient to base our notion of reduction in a core (still high-level) annotation-free language, which essentially comes from enlarging standard TRS's with a *let*-construct to express local bindings. Therefore the syntax of the core language is the same used in the *ct-let*-rewriting of Sect. 3.2.2, and so no recursive lets is allowed. Programs $\mathcal{P} \in Program_{let}$ in the core language are CS's with extra variables, with the exception that right-hand sides of program rules can contain *let*s, i.e., a program rule takes the form $f(t_1, \ldots, t_n) \rightarrow e$ with $f \in FS$, $t_1, \ldots, t_n$ a linear tuple of c-terms and $e \in LExp$. The `rtc` or `ctc` annotation of a function in an annotated program intends to determine its behavior. This is replaced in the core language by explicit local bindings made up with *let*s. The exact behavior of *let*s is given by the *rt-let*-rewriting relation defined in the next subsection, but the intuition is clear: in the reduction of $let\ X = e_1\ in\ e_2$, all occurrences of X in e_2 will share the same value, that will come from the evaluation of e_1. This gives the hint for the mapping $\tau : \mathcal{P}_{rc} \mapsto Program_{let}$ that transforms annotated programs into core programs.

Definition 3.3.3 (Sharing transformation τ, [LRS09a] Def. 1)**.**
Given a program rule $R \equiv f(\bar{t}) \rightarrow e$, its transformed rule is:

- $\tau(R) \equiv R$, if $f \in FS_{rtc}$.
- $\tau(R) \equiv f(\bar{t}) \rightarrow let\ \overline{Y = X}\ in\ e[\overline{X/Y}]$, where $FV(e) = \overline{X}$ and $\overline{Y}$ is a linear tuple of fresh variables, if $f \in FS_{ctc}$.

The auxiliary relation $\rightarrow^{rt'}$ is defined by the rules:

FApp	$f(\bar{t})\sigma \rightarrow^{rt'} e\sigma$	if $f(\bar{t}) \rightarrow e$ is a rule of $\mathcal{P}$, $\sigma \in LSubst$
RBind	$let\ X = t\ in\ e \rightarrow^{rt'} e[X/t]$	
Elim	$let\ X = e_1\ in\ e_2 \rightarrow^{rt'} e_2$	if $X \notin FV(e_2)$
Flat$_1$	$h(\ldots, let\ X = e_1\ in\ e_2, \ldots) \rightarrow^{rt'} let\ X = e_1\ in\ h(\ldots, e_2, \ldots)$	
Flat$_2$	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{rt'} let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$	
LetIn	$let\ X = \mathcal{C}[e]\ in\ e' \rightarrow^{rt'} let\ Y = e\ in\ let\ X = \mathcal{C}[Y]\ in\ e'$ where Y is fresh, if $\mathcal{C} \neq [\,]$ is a c-context and $e \equiv f(\bar{e})$ or $e \in \mathcal{V}$.	

The relation $\rightarrow^{rt}$ is defined as: for any $\mathcal{C} \in Cntx$, $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$ if $e \rightarrow^{rt'} e'$ and the following conditions hold:

i) If $e \rightarrow^{rt'} e'$ is $f(\bar{t})\sigma \rightarrow^{rt'} r\sigma$ by FApp using $(f(\bar{t}) \rightarrow r) \in \mathcal{P}$ and $\sigma \in LSubst$, then $vran(\sigma|_{\setminus var(\bar{t})}) \cap BV(\mathcal{C}) = \emptyset$

ii) If $e \rightarrow^{rt'} e'$ is $let\ X = t\ in\ a \rightarrow^{rt'} a[X/t]$ by RBind, then $var(t) \subseteq BV(\mathcal{C})$

iii) If $e \rightarrow^{rt'} e'$ is $let\ X = \mathcal{C}'[Y]\ in\ a \rightarrow^{rt'} let\ Z = Y\ in\ let\ X = \mathcal{C}'[Z]\ in\ a$ by a LetIn step, then $Y \notin BV(\mathcal{C})$

Figure 3.19: Run-time *let* rewriting relation $\rightarrow^{rt}$ [LRS09a]

The transformation is naturally extended to programs as $\tau((\mathcal{P}, FS_{rtc}, FS_{ctc})) = \{\tau(R) | R \in \mathcal{P}\}$

This transformation leaves untouched the rules for `rtc`-functions (even if `ctc`-functions are invoked in the right-hand side), and introduces a *let*-binding for each variable in the right-hand side, in the case of program rules for `ctc`-functions. For instance, the transformed rule for the `ctc`-function `palAux` in Ex. 3.3.5 will be

```
palAux(X) → let Y = X in Y ++ reverse(Y).
```

Later we will see that, for pure call-time choice programs, the behavior resulting of this transformation together with the definition of *rt-let*-rewriting given below corresponds exactly to the standard well-established semantics of call-time choice [GHLR99, LRS07b].

Semantics of the core language: *rt-let*-rewriting Reduction in the core language will consist in a careful combination of ordinary rewriting –to cope with run-time choice– and *let*-management –to express sharing and call-time choice–. The notion of *one step of reduction*, is given by the run-time rewriting relation with local bindings (or *rt-let*-rewriting), written $\rightarrow^{rt}$ (or $\rightarrow^{rt}_{\mathcal{P}}$ if the program $\mathcal{P}$ is made explicit). In order to define it we first need a little more terminology. The set $BV(\mathcal{C})$ of *bound variables of a context* consists only of those *let*-bound variables visible from the hole of $\mathcal{C}$, so it is defined by $BV([\,]) = \emptyset$; $BV(h(\ldots, \mathcal{C}, \ldots)) = BV(\mathcal{C})$; $BV(let\ X = e\ in\ \mathcal{C}) = \{X\} \cup BV(\mathcal{C})$; $BV(let\ X = \mathcal{C}\ in\ e) = BV(\mathcal{C})$. We will also employ the notion of *c-contexts*, which are contexts whose holes appear only within a nested application of constructor symbols, that is, $\mathcal{C} ::= [\,] \mid c(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $c \in CS^n$, $e_1, \ldots, e_n \in LExp$.

Now in Fig. 3.19 we can present the rules of $\rightarrow^{rt}$. There $\mathcal{P} \in Program_{let}$, $X, Y, Z \in \mathcal{V}$, $f \in FS, h \in FS \cup CS, t \in CTerm, e, e_i, a \in LExp$, and $\mathcal{C}, \mathcal{C}' \in Cntx$. Rule **FApp** allows us to perform ordinary rewriting steps: when an expression matches the left-hand side of a program rule we can replace this expression with the right-hand side of the corresponding

rule instance. Condition *i)* is imposed to avoid the capture of free extra variables introduced by σ. The rest of the $\rightarrow^{rt}$-rules forget about the program and deal only with *let*-bindings. An important intuition is that if a step $e \rightarrow^{rt'} e'$ is performed using any of these rules that are independent from the program, then the set of $\rightarrow^{rt}$-reachable values (i.e. constructor terms) will be the same for e and e'. Therefore all non-determinism involved in these rules is *don't care*; only FApp is *don't know*.

When the defining expression of a *let*-binding has been reduced to a value then the rule **RBind** (restricted bind) can be used to propagate this value to the body of the *let*, by applying the corresponding substitution. The restriction expressed in condition *ii*) is needed to be coherent with the fact that in $\rightarrow^{rt}$ we use *LSubst* for parameter passing, and so any variable can be potentially instantiated with a *LExp*. Now, notice that if we dropped condition *ii)*, a step like $let\ Y = X\ in\ (Y,Y) \rightarrow^{rt} (X,X)$ would be allowed; however, some of its particular cases (replacing the free variable X by concrete expressions) are not valid, as happens for instance with $let\ Y = word\ in\ (Y,Y) \rightarrow^{rt} (word, word)$, which is forbidden because it does not respect sharing. The property that any reduction step performed from an expression is also possible with any of its instances (obtained by a substitution of the kind allowed in parameter passing) is a desirable property, for it is very useful to reason about the programs. For example replacing the program rule $(f(X) \rightarrow let\ Y = X\ in\ (Y,Y))$ with $(f(X) \rightarrow (X,X))$ is unsound, because they provide different levels of sharing: this could be easily detected in our setting because the step $let\ Y = X\ in\ (Y,Y) \rightarrow^{rt} (X,X)$ is forbidden. **Elim** erases a *let*-binding when the bound variable does not appear in the body. The flattening rules $\mathbf{FLAT_1}$ and $\mathbf{FLAT_2}$ distribute the bindings, preventing derivations to become wrongly blocked. We remark that our variable convention ensures that application of FLAT_1 or FLAT_2 does not capture variables. The rule **LetIn** is designed to introduce *let*s only for expressions which are already shared, that is, which are present in a defining expression: introducing *let*s in more occasions would reduce the set of reachable values, causing incompleteness. Besides that, the context in which they appear must be a c-context because these LetIn steps are performed in order to enable a future RBind step, to propagate the partial value for the defining expression computed so far; the condition $C \neq [\ ]$ avoids successive and useless applications of these rules. Specifically, the case $e \in \mathcal{V}$ in rule LetIn is needed to proceed in derivations blocked by the restrictions in RBind, as illustrated by the program $\mathcal{P} = \{f(c(X)) \rightarrow true\}$ and the expression $let\ Y = c(X)\ in\ f(Y)$, to which RBind cannot be applied because X is free and therefore does not fulfil condition *ii)*. Without the case $e \in \mathcal{V}$ in LetIn, that expression would be a normal form representing incorrectly a failed computation; but using LetIn as it is proposed we can do $let\ Y = c(X)\ in\ f(Y) \rightarrow^{rt} let\ Z = X\ in\ let\ Y = c(Z)\ in\ f(Y)$; now the computation can proceed successfully by applying RBind,FApp,Elim yielding $let\ Z = X\ in\ f(c(Z)) \rightarrow^{rt} let\ Z = X\ in\ true \rightarrow^{rt} true$. The condition *iii*) affecting rule LetIn is only imposed to forbid useless steps of extraction of a bound variable, which are not needed to enable the application of RBind.

Example 3.3.6. Consider the program $\mathcal{P} = \{coin \rightarrow 0, coin \rightarrow s(0), 0 + X \rightarrow X, s(X) + Y \rightarrow s(X + Y), pos(s(X)) \rightarrow true, double(X) \rightarrow let\ Y = X\ in\ Y + Y\}$ defining some easy operations for natural numbers (represented with 0 and s in the standard way). Notice the *let*-binding in the function `double`; due to its presence, $double(coin)$ can be evaluated to 0 or $s(s(0))$, but not to $s(0)$ (that could be obtained with $\rightarrow^{rt}$ if the binding were not present). The following is a possible $\rightarrow^{rt}$-derivation with $\mathcal{P}$ for the expression $pos(double(double(coin)))$.

$$
\begin{array}{ll}
pos(\underline{double(double(coin))}) & \\
\to^{rt} \underline{pos(let\ Y = double(coin)\ in\ Y + Y)} & \textsc{Fapp} \\
\to^{rt} \underline{let\ Y = double(coin)\ in\ pos(Y + Y)} & \textsc{Flat}_1 \\
\to^{rt} \underline{let\ Y = (let\ Z = coin\ in\ Z + Z)\ in\ pos(Y + Y)} & \textsc{Fapp} \\
\to^{rt} \underline{let\ Z = coin\ in\ let\ Y = Z + Z\ in\ pos(Y + Y)} & \textsc{Flat}_2 \\
\to^{rt} \underline{let\ Z = s(0)\ in\ let\ Y = Z + Z\ in\ pos(Y + Y)} & \textsc{Fapp} \\
\to^{rt} \underline{let\ Y = s(0) + s(0)\ in\ pos(Y + Y)} & \textsc{RBind} \\
\to^{rt} \underline{let\ Y = s(0 + s(0))\ in\ pos(Y + Y)} & \textsc{Fapp} \\
\to^{rt} let\ V = 0 + s(0)\ in\ \underline{let\ Y = s(V)\ in\ pos(Y + Y)} & \textsc{LetIn} \\
\to^{rt} let\ V = 0 + s(0)\ in\ \underline{pos(s(V) + s(V))} & \textsc{RBind} \\
\to^{rt} let\ V = 0 + s(0)\ in\ \underline{pos(s(V + s(V)))} & \textsc{Fapp} \\
\to^{rt} \underline{let\ V = 0 + s(0)\ in\ true} & \textsc{Fapp} \\
\to^{rt} true & \textsc{Elim}
\end{array}
$$

This is not the only possible derivation, nor the shortest one, but it illustrates some interesting aspects of the *rt-let*-rewriting relation, and some differences with its ancestor the *ct-let*-rewriting relation, defined in Sect. 3.2.2 (page 38). The main difference is that, as mentioned above, we cannot introduce lets as freely as in *ct-let*-rewriting, where everything was shared by default: the only sharing existing here is that explicitly specified in the *lets* in program rules and expressions, like the one introduced by the first FAPP step. On the other hand in *ct-let*-rewriting we had to introduce *lets* more frequently to enable FAPP steps, that here can be performed more easily thanks to the more general type of substitutions used in the new version of FAPP. This is why we only need a new restricted version of LETIN, which permits the eventual propagation of the outer computed part of a shared term, but still keeping the sharing information untouched. This is sometimes necessary to enable an FAPP step, as we have just seen in the derivation of the example.

We have already anticipated that FAPP is the only *rt-let*-rewriting rule whose nondeterminism is don't know, while the other rules are don't care in the sense that the set of reachable values does not change after one *rt-let*-rewriting step. The following results give a more formal characterization of those intuitions.

Proposition 3.3.7 ([LRS09a] Prop. 1)**.** The relation $\to^{rt}{}_{\setminus Fapp}$ defined by the rules of Fig. 3.19 except FAPP is terminating.

Proposition 3.3.8 ([LRS09a] Prop. 2)**.** For any $e, e' \in LExp$, if $e \to^{rt *} e'$ does not use FAPP, then $|e| = |e'|$.

A conservative extension Here we will show with technical care that our framework indeed generalizes pure run-time choice—as realized by ordinary rewriting—and pure call-time choice—as realized by the *CRWL* approach—.

Proving the first statement is fairly straightforward. First assume that in our source program every function is annotated as `rtc`. Then the transformation τ will leave the program rules untouched and no *let* will appear in the program. If *let*s do not appear in the program then every step of ordinary rewriting is a valid *rt-let*-rewriting step performed by the rule FAPP of Fig. 3.19, because the absence of *let*s implies that $BV(\mathcal{C}) = \emptyset$ for any context $\mathcal{C}$, which guarantees the condition *i)* in Fig. 3.19. Moreover, if a step $e \to^{rt} e'$ has

been performed for $e, e' \in Exp$, then the only rule which may have been applied is FAPP, and besides no *let* could have been used to instantiate the extra variables: thus the step is also an ordinary rewriting step. Therefore, we have:

Theorem 3.3.9 (*Rt-let*-rewriting extends rewriting, [LRS09a] Th. 1)**.** *If $\mathcal{P}$ is a program without lets (i.e., $\mathcal{P} \in CS's$), then:*

$$e \to_{\mathcal{P}} e' \Leftrightarrow e \to^{rt}_{\mathcal{P}} e', \text{ for any } e, e' \in Exp.$$

On the other hand comparing *rt-let*-rewriting with the call-time choice semantics provided by *CRWL* is more complicated, even having at our disposal an equivalent rewrite notion for call-time choice like the *ct-let*-rewriting relation. The point is that, as we saw above, despite their rough similarity both relations are quite different; as a matter of fact, they are incomparable step by step. For this reasons we have used the $CRWL_{let}$ logic presented in Sect. 3.2.2 (page 41) as our main tool for tackling this task.

Now assume that in our source program $\mathcal{P}$ every function is annotated as `ctc`, hence $FS_{rtc} = \emptyset$ and $FS_{ctc} = FS$; we will do so until the end of this section. Besides, when using a program from $\mathcal{P}_{rc}$ in the place of a CS we denote the use of its first component, which makes sense as that will be a CS. Then the expected property of our transformation τ is that for any $\mathcal{P} \in \mathcal{P}_{rc}$ with $FS_{rtc} = \emptyset$ we have that $\tau(\mathcal{P})$ interpreted by *rt-let*-rewriting behaves exactly as $\mathcal{P}$ under a call-time choice semantics. As usual we will decompose the problem of proving the adequacy of τ into the subproblems of proving its soundness and completeness. The following is our main soundness result.

Theorem 3.3.10 (Soundness of τ, [LRS09a] Th. 4)**.**
For any program $\mathcal{P} \in \mathcal{P}_{rc}$ with $FS_{rtc} = \emptyset$, $e \in LExp$, $t \in CTerm$,

$$e \to^{rt\,*}_{\tau(\mathcal{P})} t \;\Rightarrow\; \mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$$

This result follows easily from the combination of the following ones, the former states the adequacy of τ under the $CRWL_{let}$ logic, while the latter shows that *rt-let*-rewriting steps respect call-time choice for programs transformed by τ whenever $FS_{rtc} = \emptyset$.

Theorem 3.3.11 (Adequacy of τ under $CRWL_{let}$, [LRS09a] Th. 2)**.**
For any program $\mathcal{P} \in \mathcal{P}_{rc}$, $e \in LExp$ we have $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\tau(\mathcal{P})}$. In particular, $[e]^{\mathcal{P}} = [e]^{\tau(\mathcal{P})}$.

Theorem 3.3.12 ([LRS09a] Th. 3)**.** *For any program $\mathcal{P} \in \mathcal{P}_{rc}$ with $FS_{rtc} = \emptyset$, $e, e' \in LExp$,*

$$e \to^{rt\,*}_{\tau(\mathcal{P})} e' \;\Rightarrow\; [\![e']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$$

The next goal is proving *completeness* of the simulation, i.e., the reciprocal of Th. 3.3.10. The technical key for it is the following result, ensuring that any value in the $CRWL_{let}$-semantics of an expression e can be covered by a $\to^{rt}$ derivation starting from e.

Lemma 3.3.7 (Completeness lemma for $\to^{rt}$, [LRS09a] Lemma 2)**.** *For any $\mathcal{P} \in Program_{let}$, $e \in LExp, t \in CTerm_{\perp}$,*

$$\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \;\Rightarrow\; e \to^{rt\,*}_{\mathcal{P}} e'$$

for some $e' \in LExp$ such that $t \sqsubseteq |e'|$.

Notice that the lemma, being a completeness result, does not mention the transformed program, and therefore constitutes a formal proof of the intuitive fact that the $CRWL_{let}$-semantics, designed to express call-time choice, cannot give more results than the more liberal *rt-let*-rewriting, a result which is interesting in itself.

If we apply Lemma 3.3.7 to $t \in CTerm$ (i.e., t is total), then $t \sqsubseteq |e'|$ means $t \equiv |e'|$, which in particular implies that there is no function application in $|e'|$. One could expect then that the *let*-bindings that could remain in e' could be eliminated by some $\rightarrow^{rt}$-steps, and therefore that for t total $\mathcal{P} \vdash e \twoheadrightarrow t$ implies $e \rightarrow^{rt}_{\mathcal{P}}{}^{*} t$. However, this cannot be guaranteed for total but not ground t, because a variable X in t, which is free, can appear in e' inside a *let*-binding $let\ Y = X\ in\ \ldots$ that cannot be dropped off because of the condition *ii)* imposed to $\rightarrow^{rt}$ in Fig. 3.19. Which can be proved is the following:

Theorem 3.3.13 (Completeness of $\rightarrow^{rt}$ wrt. $CRWL_{let}$, [LRS09a] Th. 5)**.**
For any $\mathcal{P} \in Program$, $e \in LExp$, and $t \in CTerm$,

$$\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \quad \Rightarrow \quad e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = X}\ in\ t'$$

for some $t' \in CTerm$ such that $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$.
If in addition t is ground, then $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$.

Joining all these completeness results with the previous soundness results and the equivalence of $\mathcal{P}$ and $\tau(\mathcal{P})$ wrt. $CRWL_{let}$, it is not difficult now to obtain the adequacy (soundness + completeness) of the transformation τ to express call-time choice under an overall run-time choice regime.

Theorem 3.3.14 (Adequacy of τ, [LRS09a] Th. 6)**.**
For any $\mathcal{P} \in \mathcal{P}_{rc}$ with $FS_{rtc} = \emptyset$, $e \in LExp$, $t \in CTerm_{\perp}$,

a) $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \ \Leftrightarrow\ e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e'$, for some e' such that $|e'| \sqsupseteq t$.

b) If $t \in CTerm$ (i.e., t is total), then:

$$\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \ \Leftrightarrow\ e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = X}\ in\ t'$$

for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$.

c) If t is total and ground, then $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t \Leftrightarrow e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$.

Combining Call-time and Run-time Choice in a Run-time Choice Environment: conclusions We refer the reader to sections 3.3, 5 and 6 of [LRS09a] for a discussion about related work and some other possible design alternatives. Our proposal starts from a run-time choice framework and extends it by allowing the combination of call-time and run-time choice either at the level of CS's with annotated functions, or at the lower level of *rt-let*-rewriting. We have already seen some examples of the former (e.g. Ex. 3.3.5 (page 79), see Sect. 2 of [LRS09a] for more examples), and we think that the latter can be useful not only for programming purposes, but also for devising and justifying in a formal basis program transformations or implementation techniques. As an example consider the function `repeat'`, similar to `star` of Ex. 3.3.5, but in this case programmed to follow call-time choice:

```
repeat'(X) → let Y=X in [Y|repeat'(Y)]
```

With this definition, an expression of the form `repeat'(e)` reduces to the expression `let Y=e in [Y|repeat'(Y)]`, and therefore recursive invocations to `repeat'` (and there might be an arbitrarily large number of them in a lazy computation) generate successive *let*-bindings `let Z=Y in [Z|repeat'(Z)]`, etc. However, intuitively only the first `let Y=e` is really needed, since then Y is already a shared value for which new sharing is useless. This suggests (automatically) replacing the original definition of `repeat'` by an optimized variant

```
repeat'(X) → let Y=X in [Y|repeat(Y)]
repeat(X) → [X|repeat(X)]
```

that does not need to employ useless *let*s. We see some analogy between these *let*-binding savings described here and the implementation of sharing in some *Curry* systems [AH00] that try to avoid unnecessary creation of *suspensions*. A thorough investigation of these issues is left for future work. We simply remark here the potential applicability of our framework as a suitable formalism for making and proving precise statements.

A prototype implementation of this framework based on the *Toy* system has been developed [LRS09e], and it is publicly available. It is a simple modification of *Toy* and thus its performance could be improved a lot, but it is still useful for experimenting with this new framework. Besides, it enjoys some extra features of *Toy* not covered by our framework, like the use of higher order functions. These are particularly interesting, because we can use them to make the management of call-time choice more modular and abstract through a HO polymorphic function `call_time F X → let Y = X in F Y`. With this function (that can be generalized to greater arities, simply by iterated composition) we can get call-time versions of functions following other regimes. These higher order capabilities can also be employed to solve the problems of using the classical transformational technique of [WB89] to implement type classes in FLP pointed out by Lux in [Lux09]. The interested reader can find a proposal of solution in [Rod09].

In a call-time choice environment

In [LRS09c] (Sect. 7.2.4, page 140) we can find another approach to the combination of call-time and run-time choice parameter passing, this time starting from the call-time choice framework of *CRWL*. This framework is extended to provide support for run-time choice in localized parts of a program, by means of some primitives to "unshare" expressions thus allowing to make copies of them whose evaluation is independent. The extension is remarkably simple at three relevant levels: syntax, formal operational calculi and implementation, which is based on the system *Toy*.

Again we work at different abstraction levels. User programs are CS's with extra variables but where the syntax of expressions has been extended with two primitives $rt(\_)$ and $rRt(\_)$ to express run-time choice in different ways. User programs are then translated into a core language in which function symbols may be optionally annotated with a rt superscript, indicating that those function symbols will be treated as a constructor symbol as far sharing and parameter passing is concerned. We have given two different semantic formulations for these core programs, the former is an extension of *CRWL*-rewriting as defined in Fig. 3.10 (page 39), while the latter extends the *let*-rewriting relation of Fig. 3.21 (page 88). We have also given a semantic formulation for user programs using the $rRt(\_)$ primitive by means of an extension of the *CRWL* logic.

The rt primitive We start from the framework of *CRWL*, therefore the sets Exp and $CTerm$ of expressions and c-terms are defined like in Sect. 3.1.1. By introducing the rt primitive two new sets are defined, the set of expressions with run-time choice annotations $RtExpr \ni e ::= X \mid c(e_1,\ldots,e_n) \mid f(e_1,\ldots,e_n) \mid rt(e)$ where $e_1,\ldots,e_m,e \in RtExpr$; and the set of $RtCTerm$ of annotated c-terms, $RtCTerm \ni t ::= X \mid c(t_1,\ldots,t_n) \mid rt(e)$ where $t_1,\ldots,t_m \in RtCTerm$ and $e \in RtExpr$. Then user programs are CS's with extra variables where the set Exp has been replaced by $RtExpr$, i.e., set of program rules of the shape $f(t_1,\ldots,t_n) \to e$ where $(t_1,\ldots,t_n)$ is a linear tuple of c-terms from $CTerm$, and $e \in RtExpr$.

Now we will explain the meaning of the rt primitive is the following. When we apply a program rule $f(t_1,\ldots,t_n) \to e$ to an expression $a \equiv f(t_1\theta,\ldots,t_n\theta)$ this expression is reduced to $r\theta$, but under the following informal criterion about *sharing*: the copies of any subexpression b of a created in $r\theta$ are not shared –i.e. follow run-time choice– if b is in a position below an application of the rt primitive, and shared –i.e. follow call-time choice– otherwise. These ideas are formalized in the next section.

The core language and its semantics Instead of giving a semantics for annotations $rt(e)$ directly, we think about it as a syntactic sugar for the annotation of the function symbols that appear in e with a rt superscript, indicating that those function symbols will be treated as a constructor symbol as far sharing and parameter passing is concerned. Therefore, an expression containing only variables, constructor symbols and function symbols annotated with rt could be copied freely, thus getting a run-time behaviour for it, as a function argument. We write FS^{rt} for the set of function symbols with superscript rt, $FS^?$ for $FS \cup FS^{rt}$ and $f^?$ for function symbols in $FS^?$, i.e., for possibly superscripted function symbols.

The desugaring of expressions to eliminate the rt primitive transforming it into rt annotations is performed according to the following definition:

Definition 3.3.4 (Desugaring of the rt primitive, [LRS09c] Def. 1)**.**

$$\begin{array}{lcll} desugar(rt(X)) & = & X & \text{if } X \in \mathcal{V} \\ desugar(rt(c(e_1,\ldots,e_n))) & = & c(desugar(rt(e_1)),\ldots,desugar(rt(e_n))) & \text{if } c \in CS \\ desugar(rt(f(e_1,\ldots,e_n))) & = & f^{rt}(desugar(rt(e_1)),\ldots,desugar(rt(e_n))) & \text{if } f \in FS \\ desugar(rt(rt(e))) & = & desugar(rt(e)) & \end{array}$$

According to this syntactic desugaring for $rt(e)$, the syntax of annotated c-terms and expressions can be reformulated as follows:

- $RtCTerm \ni t ::= X \mid c(t_1,\ldots,t_n) \mid f^{rt}(t_1,\ldots,t_n)$, if $X \in \mathcal{V}$, $c \in CS^n$, $f \in FS^n$, $t_1,\ldots,t_n \in RtCTerm$
- $RtExpr \ni e ::= X \mid c(e_1,\ldots,e_n) \mid f^?(e_1,\ldots,e_n)$, if $X \in \mathcal{V}$, $c \in CS^n$, $f^? \in FS^?$, $e_1,\ldots,e_n \in RtExpr$

To express parameter passing in function applications with rt-annotated arguments we will need to consider rt-c-substitutions, defined by: $\theta \in RtCSubst$ iff $X\theta \in RtCTerm, \forall X \in \mathcal{V}$.

Now we will define calculi to work with annotated expressions. We will extend both *CRWL*-rewriting (see Fig. 3.10 (page 39)) and *let*-rewriting (see Fig. 3.21 (page 88)) to get two (hopefully) equivalent characterizations of a semantics for annotated run-time choice

B	$\mathcal{C}[e] \rightarrowtail \mathcal{C}[\bot]$	for any context $\mathcal{C}$ and expression $e \in RtExpr_\bot$
OR	$\mathcal{C}[f^?(\overline{p})\theta] \rightarrowtail \mathcal{C}[r\theta]$	for any context $\mathcal{C}$, $(f(\overline{p}) \rightarrow r \in)\mathcal{P}$, and $\theta \in RtCSubst_\bot$

Figure 3.20: *CRWL*-rewriting with rt annotations [LRS09c]

FAPP $f^?(\overline{p})\theta \rightarrow_l r\theta$, if $(f(\overline{p}) \rightarrow r) \in \mathcal{P}$, $\theta \in RtCSubst$

LETIN $h(\ldots, e, \ldots) \rightarrow_l let\ X = e\ in\ h(\ldots, X, \ldots)$, if $h \in \Sigma$, $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable

BIND $let\ X = t\ in\ e \rightarrow_l e[X/t]$, if $t \in RtCTerm$

ELIM $let\ X = e_1\ in\ e_2 \rightarrow_l e_2$, if $X \notin FV(e_2)$

FLAT $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

CONTX $\mathcal{C}[e] \rightarrow_l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrow_l e'$ using any of the previous rules, and in case $e \rightarrow_l e'$ is a FAPP step using $(f(\overline{p}) \rightarrow r)\theta \in [\mathcal{P}]$ then $vran(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$.

Figure 3.21: Rules of *let*-rewriting extended with rt annotations [LRS09c]

under a call-time choice environment. This extensions are formulated in Fig. 3.20 and Fig. 3.21, and are just simple extensions of their corresponding ancestor relations that treat symbols from FS^{rt} like constructor symbols.

Example 3.3.7 ([LRS09c] Ex. 1)**.** Given the program

$$\begin{array}{ll} coin \rightarrow 0 & f(X) \rightarrow g(X, coin) \\ coin \rightarrow 1 & g(X, Y) \rightarrow (X, X, Y, Y) \end{array}$$

we want to evaluate the expression $rt(f(coin))$, which is desugared as $f^{rt}(coin^{rt})$. With the calculus of Fig. 3.20 we can do:

$$\begin{array}{l} f^{rt}(coin^{rt}) \rightarrowtail g(coin^{rt}, coin) \rightarrowtail g(coin^{rt}, 0) \rightarrowtail (coin^{rt}, coin^{rt}, 0, 0) \\ \rightarrowtail (0, coin^{rt}, 0, 0) \rightarrowtail (0, 1, 0, 0) \end{array}$$

Note how in the first step the expression $f^{rt}(coin^{rt})$ can be evaluated as every function symbol present in $coin^{rt}$ is annotated with rt. On the other hand we cannot apply (OR) to $g(coin^{rt}, coin)$, as one of its arguments contains a function symbol that it is not annoted for run-time, and thus the value $(0, 1, 0, 1)$ is not reachable from $f^{rt}(coin^{rt})$. This is even more evident in the version of this evaluation got with the calculus of Fig. 3.21:

$$\begin{array}{l} f^{rt}(coin^{rt}) \rightarrow_l g(coin^{rt}, coin) \rightarrow_l let\ X = coin\ in\ g(coin^{rt}, X) \\ \rightarrow_l let\ X = coin\ in\ (coin^{rt}, coin^{rt}, X, X) \rightarrow_l let\ X = coin\ in\ (0, coin^{rt}, X, X) \\ \rightarrow_l let\ X = coin\ in\ (0, 1, X, X) \rightarrow_l let\ X = 0\ in\ (0, 1, X, X) \\ \rightarrow_l (0, 1, 0, 0) \end{array}$$

When we reach the expression $let\ X = coin\ in\ (coin^{rt}, coin^{rt}, X, X)$ it is clear that the first two components of the tuple may evolve in different ways while the values of the last two components will be shared.

The rRt primitive Many other primitives to express run-time choice can be conceived, here we will present another alternative, the rRt primitive, whose behaviour is defined by the he following inference rule that should be added to the *CRWL* logic:

$$\frac{e \to^*_{\mathcal{P}'} e' \quad t \sqsubseteq |e'|}{\mathcal{P} \vdash_{CRWL} rRt(e) \twoheadrightarrow t} \ \text{RRT}$$

where $\mathcal{P}'$ is the program resulting of adding to $\mathcal{P}$ the new rule $rRt(e) \to e$. The rule RRT itself is already suggesting a possible implementation for rRt. This implementation will be based on the fact that, for any program in which every function symbol that appears in a right hand side of a program rule is rt-annotated, the evaluation of an expression that has each of its function symbols rt-annotated too returns the same results as it was evaluated under run-time choice but discarding the annotations. This ideas are formalized in the following definition:

Definition 3.3.5 ([LRS09c] Def. 2)**.** Given a *CRWL*-program $\mathcal{P}$:

- We build the signature of a new program $\_\mathcal{P}$ adding to it any constructor symbol in the signature of $\mathcal{P}$, and for any function symbol f in the signature of $\mathcal{P}$ considering a fresh function symbol $\_f$ which we add to the signature of $\_\mathcal{P}$.
- We define the transformation of expressions rRt as:

$$\begin{array}{lcll} rRtT(X) & = & X & \text{if } X \in \mathcal{V} \\ rRtT(c(e_1,\ldots,e_n)) & = & c(rRtT(e_1),\ldots,rRtT(e_n)) & \text{if } c \in CS \\ rRtT(f(e_1,\ldots,e_n)) & = & \_f^{rt}(rRtT(e_1),\ldots,rRtT(e_n)) & \text{if } f \in FS \\ rRtT(rRtT(e)) & = & rRtT(e) & \end{array}$$

- For any $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ we add the rule $\_f(p_1,\ldots,p_n) \to rRtT(r)$ to $\_\mathcal{P}$.

Finally, any expression $rRt(e)$ to be evaluated under $\mathcal{P}$ is desugared into $rRtT(e)$ and evaluated under $\mathcal{P} \uplus \_\mathcal{P}$

Example 3.3.8 ([LRS09c] Ex. 2)**.** Starting with the program of Example 3.3.7 we get the program

$$\begin{array}{c} \{coin \to 0, coin \to 1, f(X) \to g(X, coin), g(X,Y) \to (X,X,Y,Y)\} \\ \uplus \\ \{\_coin \to 0, \_coin \to 1, \_f(X) \to \_g^{rt}(X, coin), \_g(X,Y) \to (X,X,Y,Y)\} \end{array}$$

under which we can do:

$$\begin{array}{l} rRt(f(coin)) \equiv \_f^{rt}(\_coin^{rt}) \rightarrowtail \_g^{rt}(\_coin^{rt}, \_coin^{rt}) \\ \rightarrowtail (\_coin^{rt}, \_coin^{rt}, \_coin^{rt}, \_coin^{rt}) \rightarrowtail^* (0,1,0,1) \end{array}$$

Implementation issues In order to study the practicability of the proposal we have implemented the rt primitive [LRS08b] as an extension of *Toy*. This system, as well as other modern systems like *Curry*, operates under call-time choice, so the introduction of a new syntactic construct $rt\ e$ is what enables the use of run-time choice in some parts of the program. Again note that the resulting prototype enjoys some extra features of *Toy* not modelled by the presented semantics, like its higher order capabilities.

The extension is well supported by the system and requires only some lightweight modifications. In fact, the traditional problem is how to achieve sharing in a non-deterministic language like this, and our goal now is to inhibit this sharing mechanism at the points required by the programmer with *rt*. *Toy* is implemented in Prolog and uses Prolog as target code (see [LLR93, CSe06] for details). Sharing is implemented by means of *suspensions*, that are Prolog terms of the form:

susp(FunctionName,Arguments,Result,Evaluated)

The *FunctionName* and its *Arguments* represent the expression e to be evaluated, while *Result* is the resulting value (if evaluated, variable in other case) and *Evaluated* is a flag that indicates if the expression has been evaluated (flag *on*) or not (flag variable). Every function call is translated into a suspension in order to share its value when the expression is passed as argument and copied. As an example of the use of this representation consider the following program:

```
coin = 0
coin = 1

double X = X + X

test1 = double coin
test2 = rt (double coin)
```

Consider the evaluation of *test1*. As all the function calls are translated into suspended forms, in particular *coin* will have the form *susp(coin,[],R,E)*. The evaluation of *double* does not demand the evaluation of its argument *coin*, so it will produce

susp(coin,[],R,E) + susp(coin,[],R,E)

Later, when one of the calls to *coin* is evaluated, for example to 0, the other one automatically gets the same value:

susp(coin,[],0,on) + susp(coin,[],0,on)

The result of the addition is 0, that is a value obtained for *test1*. If we evaluate *coin* to 1 we have

susp(coin,[],1,on) + susp(coin,[],1,on)

and then the result 2, that is the other value obtained for *test1*. With this sharing mechanism we can not obtain the value 1 for *double coin* as it would require to evaluate both calls to *coin* to two different values.

For the function function *test2* we would want to obtain the values 0 and 2 as before, but also the value 1 (evaluating separately both calls to *coin*). In this case *rt* will deactivate the sharing mechanism. This can be easily achieved by translating the call *coin* into the suspended form *susp(coin,[],R,rt)*. The flag *rt* will indicate to the system that the value of this expression must not be shared (and neither kept in the variable R). For *test2* we evaluate

susp(coin,[],R,rt) + susp(coin,[],R,rt)

The first suspension can be reduced to 0 (without annotating the result in R), and the second one to 1, obtaining 1 for *test2* as expected.

The extension implemented in *Toy* provides this behaviour with *test1* and *test2*. In fact, for *test2* it obtains 0, 2 and 1 twice (evaluating the first *coin* to 0 and the second to 1 and vice versa). As another example, consider the problem of generating numbers as combinations of the digits 0, 1 and 2. Using the standard Haskell [PJ03] functions *take* and *repeat*, and the alternative operator '|' we could define:

```
number N = take N (repeat (0 | 1 | 2))
```

but then the expression *number 3* will produce only the answers [0,0,0], [1,1,1] and [2,2,2], because the expression *0 | 1 | 2* is evaluated only once and then its value is shared when evaluating *repeat*. For achieving the expected behaviour we have to instruct the system for choosing the digits under run-time choice (to avoid sharing):

```
number N = take N (repeat (rt (0 | 1 | 2)))
```

Now we obtain the 27 possible combinations that include $[1, 1, 2]$ or $[3, 1, 2]$ as instance. The example of palindromes of Ex. 3.3.5 (page 79) can also be simulated within this framework.

Combining Call-time and Run-time Choice in a Call-time Choice Environment: conclusions In this section we presented a simple way of combining in the same program run-time choice and call-time choice, by extending the call-time choice framework of *CRWL*. We have proposed two variants of this idea, the first being more 'local' in its effect ($rt(\_)$ primitive), while the second is more global ($rRt(\_)$ primitive). In both cases we have given a formal definition of the intended semantics.

For the first variant besides giving formal operational descriptions we have obtained a prototype implementation by modifying the system *Toy*. For the second variant the transformations of Def. 3.3.5 could also be used to develop an implementation based on the first one. Anyway the first variant again is expressive enough to solve the sharing problems [Lux09] of the dictionary translation of type classes [WB89] for FLP, as the interested reader could see in [Rod09].

If we compare this approach to the run-time choice based approach of Sect. 3.3.3, we can conclude that that approach seems to be more amenable to formal treatments, as shown by the good number of technical results obtained for that framework. On the other hand, the implementation of the call-time choice based approach was pretty straightforward, specially when reusing existing call-time-choice based implementations.

3.4 Plural Semantics

"It is only proper to realize that language is largely a historical accident. The basic human languages are traditionally transmitted to us in various forms, but their very multiplicity proves that there is nothing absolute and necessary about them. Just as languages like Greek or Sanskrit are historical facts and not absolute logical necessities, it is only reasonable to assume that logics and mathematics are similarly historical, accidental forms of expression. They may have essential variants, i.e. they may exists in other forms than the ones to which we are accustomed. Indeed, the nature of the central nervous system and of the message system that it transmits indicate positively that this is so."

John von Neumann — The Computer and the Brain - 1958

3.4.1 $\pi CRWL$: A Plural Semantics for Constructor Systems

As we already mentioned in Sect. 1.1, although call-time choice parameter passing is equivalent to having a singular semantics for non-determinism, it is not the case for run-time choice and a plural semantics. The point is that the introduction of pattern matching makes the traditional identification of run-time choice with a plural semantics wrong. We can find a proof of that in Ex. 1.1.2 (page 6), which uses the pretty simple program $\mathcal{P} = \{f(c(X)) \to d(X,X), X \ ? \ Y \to X, X \ ? \ Y \to Y\}$. There such a simple pattern like $c(X)$ is enough to force a different behaviour in the evaluation of the expression $f(c(0)?c(1))$ with run-time choice and with an hypothetical plural semantics.

- Under run-time choice, that is, term rewriting, the evaluation of the choice between $c(0)$ and $c(1)$ is needed in order to get an expression matching the pattern $c(X)$. Thus $d(0,0)$ and $d(1,1)$ are correct values for $f(c(0)?c(1))$ but it is not the case for neither $d(0,1)$ nor $d(1,0)$.
- On the other hand under a plural semantics we could consider the set $\{c(0), c(1)\}$ which are values for $c(0)?c(1)$; each of those values also matches the pattern $c(X)$. After performing parameter passing with these values we would end up with the "set-expression" (similar to a s-expression like those used in Sect. 3.3.1) $d(\{0,1\}\{0,1\})$, that yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$ and $d(1,0)$.

In the present section we will present a concrete proposal for that plural semantics, as it was first presented in [Rod08] (Sect. 7.1.3, page 126). It consists of a variation of $CRWL$ called $\pi CRWL$ (where π stands for "plural") which induces a notion of denotation based on c-terms that it is still compositional, in contrast to what happens with any c-term based semantics for run-time choice, as we saw in Sect. 3.3.1 (Ex. 3.3.1 (page 67)). Going back to c-terms is interesting not only because it is the traditional notion of value used in (first order) FLP, but also because it is much simpler than the notion of $SCTerm$ from Sect. 3.3.1 needed to recover compositionality under run-time choice, in which different non-deterministic alternatives can be packaged under constructors. Using a familiar and simpler notion of value may help to make programs easier to understand by the programmer, a characteristic always desired.

Besides formulating this novel semantics we study some of its properties, give precise technical results comparing it with call-time choice and run-time choice, and give some examples of the kind of programs for which this semantics could be useful, usually programs that need to do some collecting work. We will also deal with this last matter in the next

section, where we show a transformational prototype implementation of $\pi CRWL$ on top of the Maude system [CDE+07].

The semantics

Just like *CRWL* and term rewriting, which are the main formulations we have used for call-time choice and run-time choice, and in line with the rest of the thesis, our semantics will use CS's as programs. The current results are restricted to CS's without extra variables, but we think that the results could be easily extended to programs with extra variables, which we consider a complementary subject of future work.

We also assume that every CS contains the rules $\{X\ ?\ Y \to X, X\ ?\ Y \to Y, if\ true\ then\ X \to X\}$, defining the behaviour of $\_?\_ \in FS^2$, $if\_then\_ \in FS^2$, both used in mixfix mode, and that those are the only rules for these function symbols. For the sake of conciseness we will often omit these rules when presenting a CS.

The new calculus $\pi CRWL$ is defined by modifying the rules of *CRWL*—see Fig. 3.1 (page 22)—to consider sets of partial values for parameter passing instead of single partial values: hence, only the rule OR should be modified. To avoid the need of extending the syntax with new constructions to represent those "set expressions" that we talked about above, we will exploit the fact that $[\![e_1\ ?\ e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$. An alternative approach could have used the s-expressions of Sect. 3.3.1, but they are not really needed for the formulation of $\pi CRWL$, as we will see next.

Therefore the substitutions used for parameter passing will map variables to "disjunctions of values", this is formalized in the notion of $CSubst^?_\bot$. We define the set $CSubst^?_\bot = \{\theta \in Subst_\bot \mid \forall X \in dom(\theta), \theta(X) = t_1\ ?\ \ldots\ ?\ t_n$ such that $t_1, \ldots, t_n \in CTerm_\bot, n > 0\}$, for which $CSubst_\bot \subseteq CSubst^?_\bot \subseteq Subst_\bot$ obviously holds. The operator $? : CSubst^*_\bot \to CSubst^?_\bot$ constructs the $CSubst^?_\bot$ corresponding to a non empty sequence of $CSubst_\bot$, and is defined as $?(\theta_1 \ldots \theta_n)(X) = X$ if $X \notin \bigcup_{i \in \{1,\ldots,n\}} dom(\theta_i)$; $?(\theta_1 \ldots \theta_n)(X) = \rho_1(X)\ ?\ \ldots\ ?\ \rho_m(X)$, where $\rho_1 \ldots \rho_m = \theta_1 \ldots \theta_n \mid \lambda\theta.(X \in dom(\theta))$, otherwise. Then $dom(?(\theta_1 \ldots \theta_n)) = \bigcup_i dom(\theta_i)$. This operator is overloaded to handle non empty sets $\Theta \subseteq CSubst_\bot$ as $?\Theta = ?(\theta_1 \ldots \theta_n)$ where the sequence $\theta_1 \ldots \theta_n$ corresponds to an arbitrary reordering of the elements of Θ.

The $\pi CRWL$-proof calculus is presented in Figure 3.22. The only difference with the original *CRWL* calculus is that the rule OR has been replaced by **POR** (plural outer reduction), in which we may compute more that one partial value for each argument, and then use a substitution from $CSubst^?_\bot$ instead of $CSubst_\bot$ for parameter passing, achieving a plural semantics[6]. This calculus derives reduction statements of the form $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ that express that t is (or approximates to) a possible value for e in this semantics, under the program $\mathcal{P}$. The *$\pi CRWL$-denotation* of an expression $e \in Exp_\bot$ under a program $\mathcal{P}$ in $\pi CRWL$ is defined as $[\![e]\!]^{pl}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t\}$.

Example 3.4.1 ([Rod08] Ex. 3)**.** Consider the program of Ex. 1.1.1 (page 5), that is $\{f(c(X)) \to d(X, X),\ X\ ?\ Y \to X,\ X\ ?\ Y \to Y\}$. The following is a $\pi CRWL$-proof for the statement $f(c(0)?c(1)) \twoheadrightarrow d(0, 1)$ (some steps have been omitted for the sake of conciseness):

[6] In fact angelic non-strict plural non-determinism.

$$\textbf{RR}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad \textbf{DC}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\quad c \in CS^n$$

$$\textbf{B}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{POR}\ \frac{\begin{array}{ccccc} e_1 \twoheadrightarrow p_1\theta_{11} & & & & e_n \twoheadrightarrow p_n\theta_{n1} \\ \ldots & & \ldots & & \ldots \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & & & e_n \twoheadrightarrow p_n\theta_{nm_n} \quad r\theta \twoheadrightarrow t \end{array}}{f(e_1,\ldots,e_n) \twoheadrightarrow t}$$

$$(f(\overline{p}) \to r) \in \mathcal{P},\ \theta = ?\{\theta_{11},\ldots,\theta_{1m_1}\} \uplus \ldots \uplus ?\{\theta_{n1},\ldots,\theta_{nm_n}\}$$
$$\forall i,j\ \theta_{ij} \in CSubst_\bot \wedge dom(\theta_{ij}) = var(p_i), \forall i\ m_i > 0$$

Figure 3.22: Rules of $\pi CRWL$ [Rod08]

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \twoheadrightarrow 0}\,\text{DC}}{c(0) \twoheadrightarrow c(0)}\,\text{DC} \quad \dfrac{}{c(1) \twoheadrightarrow \bot}\,\text{B} \quad c(0) \twoheadrightarrow c(0)}{c(0)?c(1) \twoheadrightarrow c(0)}\,\text{POR} \quad c(0)?c(1) \twoheadrightarrow c(1) \quad \dfrac{0?1 \twoheadrightarrow 0 \quad 0?1 \twoheadrightarrow 1}{d(0?1,0?1) \twoheadrightarrow d(0,1)}\,\text{DC}}{f(c(0)?c(1)) \twoheadrightarrow d(0,1)}\,\text{POR}$$

Some properties of $\pi CRWL$

$\pi CRWL$ enjoys some nice properties, like the following monotonicity property, where for any proof we define its *size* as the number of applications of rules of the calculus.

Lemma 3.4.1 (Monotonicity of $\pi CRWL$, [Rod08] Lemma 4)**.** *For any CRWL-program, $e, e' \in Exp_\bot$, $t, t' \in CTerm_\bot$ if $e \sqsubseteq e'$ and $t' \sqsubseteq t$ then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ implies $\mathcal{P} \vdash_{\pi CRWL} e' \twoheadrightarrow t'$ with a proof of the same or smaller size.*

One of the most important properties is its compositionality, a property inherited from $CRWL$ which is apparent just looking at the rule of $\pi CRWL$.

Theorem 3.4.1 (Compositionality of $\pi CRWL$, [Rod08] Th. 5)**.** *For any CRWL-program, $\mathcal{C} \in Contx$ and $e \in Exp_\bot$, $[\![\mathcal{C}[e]]\!]^{pl} = \bigcup_{\{t_1,\ldots,t_n\} \subseteq [\![e]\!]^{pl}} [\![\mathcal{C}[t_1\ ?\ \ldots\ ?\ t_n]]\!]^{pl}$, for any arrangement of the elements of $\{t_1,\ldots,t_n\}$ in $t_1\ ?\ \ldots\ ?\ t_n$.*

In [Rod08] the correctness of the bubbling property we saw for $HOCRWL_{let}$ in Th. 3.2.22 from Sect. 3.2.4 (page 56) was stated for $\pi CRWL$ as Th. 6. Nevertheless that statement is false and an erratum in the text, as the following counterexample shows.

Counterexample 3.4.1. Consider the program $\mathcal{P} = \{pair(X) \to (X,X), X\ ?\ Y \to X, X\ ?\ Y \to Y\}$ and the expressions $pair(0\ ?\ 1)$ and $pair(0)\ ?\ pair(1)$ which correspond to a bubbling step using $\mathcal{C} = pair([])$. It is easy to check that $(0,1) \in [\![pair(0\ ?\ 1)]\!]^{pl}$ while $(0,1) \notin [\![pair(0)\ ?\ pair(1)]\!]^{pl}$.

Although this is an important erratum, its impact in the rest of the work is negligible, as it is not used in the proof of any of the other results.

$\pi CRWL$ also has some monotonicity properties related to substitutions. We define the preorder $\sqsubseteq_\pi$ over $CSubst^?_\bot$ by $\theta \sqsubseteq_\pi \theta'$ iff $\forall X \in \mathcal{V}$, given $\theta(X) = t_1\ ?\ \ldots\ ?\ t_n$ and $\theta(X) = t'_1\ ?\ \ldots\ ?\ t'_m$ then $\forall t \in \{t_1,\ldots,t_n\} \exists t' \in \{t'_1,\ldots,t'_m\}$ such that $t \sqsubseteq t'$; and the preorder $\trianglelefteq$ over $Subst_\bot$ by $\sigma \trianglelefteq \sigma'$ iff $\forall X \in \mathcal{V}$, $[\![\sigma(X)]\!]^{pl} \subseteq [\![\sigma'(X)]\!]^{pl}$.

Lemma 3.4.2 ([Rod08] Lemma 7). *For any CRWL-program, $e \in Exp_\bot$, $t \in CTerm_\bot$, $\sigma, \sigma' \in Subst_\bot$, $\theta, \theta' \in CSubst^?_\bot$:*

1. ***Strong monotonicity of Subst$_\bot$:*** *If $\forall X \in \mathcal{V}, s \in CTerm_\bot$ given $\mathcal{P} \vdash_{\pi CRWL} \sigma(X) \twoheadrightarrow s$ with size K we also have $\mathcal{P} \vdash_{\pi CRWL} \sigma'(X) \twoheadrightarrow s$ with size $K' \leq K$, then $\vdash_{\pi CRWL} e\sigma \twoheadrightarrow t$ with size L implies $\vdash_{\pi CRWL} e\sigma' \twoheadrightarrow t$ with size $L' \leq L$.*
2. ***Monotonicity of CSubst$_\bot$:*** *If $\theta, \theta' \in CSubst_\bot$ and $\theta \sqsubseteq \theta'$ then $\mathcal{P} \vdash_{\pi CRWL} e\theta \twoheadrightarrow t$ with size K implies $\mathcal{P} \vdash_{\pi CRWL} e\theta' \twoheadrightarrow t$ with size $K' \leq K$.*
3. ***Monotonicity of Subst$_\bot$:*** *If $\sigma \trianglelefteq \sigma'$ then $[\![e\sigma]\!]^{pl} \subseteq [\![e\sigma']\!]^{pl}$.*
4. ***Monotonicity of CSubst$^?_\bot$:*** *If $\theta \sqsubseteq_\pi \theta'$ then $[\![e\theta]\!]^{pl} \subseteq [\![e\theta']\!]^{pl}$.*

We also conjecture that for $\theta \in CSubst^?_\bot$ if $\vdash_{\pi CRWL} e \twoheadrightarrow t$ then $[\![t\theta]\!]^{pl} \subseteq [\![e\theta]\!]^{pl}$. We have not proved this result yet, and we think that an alternative formulation of $\pi CRWL$ based on s-expressions would be the right tool to make the proof of this result straightforward. All these are interesting subjects of future work.

A hierarchy of semantics for CS's

Now we will make a technical comparison between $\pi CRWL$ and the former semantics for call-time choice and run-time choice regarding the sets of c-terms computed by each semantics. This would be similar to what we did in Sect. 3.3.2, where we established that, in general, run-time choice computes more values than call-time choice. We will extend that hierarchy to place the new semantics $\pi CRWL$ in it.

As $\pi CRWL$ is a modification of $CRWL$, the relation between them is very direct.

Theorem 3.4.2 ([Rod08] Th. 9). *For any CRWL-program $\mathcal{P}$, $e \in Exp_\bot, t \in CTerm_\bot$ given a CRWL-proof for $\mathcal{P} \vdash e \twoheadrightarrow t$ we can build a πCRWL-proof for $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ just replacing every **OR** step by the corresponding **POR** step. As a consequence $[\![e]\!]^{sg}_{\mathcal{P}} \subseteq [\![e]\!]^{pl}_{\mathcal{P}}$.*

Concerning the relation of $CRWL$ and $\pi CRWL$ with term rewriting, we define the denotation of $e \in Exp$ under term rewriting as $[\![e]\!]^{rw} = \{t \in CTerm_\bot \mid \exists e' \in Exp\ .\ e \rightarrow^* e' \wedge t \sqsubseteq |e'|\}$. We can use it to recast Th. 3.3.6 (page 75) as follows:

Theorem 3.4.3 ([Rod08] Th. 10). *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rw}$. The converse inclusion does not hold in general.*

As we saw in Ex. 1.1.1, in general call-time choice semantics like $CRWL$ produce less results than run-time choice semantics like the one induced by term rewriting. We will see that this kind of relation also holds for term rewriting and $\pi CRWL$.

Theorem 3.4.4 ([Rod08] Th. 11). *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{rw} \subseteq [\![e]\!]^{pl}$. The converse inclusion does not hold in general.*

The key for proving Theorem 3.4.4 is a lemma stating that $\forall e, e' \in Exp$ if $e \rightarrow e'$ then $[\![e']\!]^{pl} \subseteq [\![e]\!]^{pl}$, that is, that every rewriting step is sound wrt. $\pi CRWL$. The evident corollary for these theorems is the announced inclusion chain.

Corollary 3.4.1 ([Rod08] Corollary 12). *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rw} \subseteq [\![e]\!]^{pl}$. Hence $\forall t \in CTerm$, $\vdash_{CRWL} e \twoheadrightarrow t$ implies $e \rightarrow^* t$ which implies $\vdash_{\pi CRWL} e \twoheadrightarrow t$.*

We conclude this section with an example of the use of $\pi CRWL$ to model problems in which some collecting work has to be done.

Example 3.4.2 ([Rod08] Ex. 8)**.** We want to represent the database of a bank in which we hold some data about its employees. This bank has several branches and we want to organize the information according to them. So we define a non-deterministic function *branches* to represent the set of branches: a set is identified then with a non-deterministic expression. In this line we define a non-deterministic function *employees* which conceptually returns the set of records containing the information regarding the employees that work in a branch. Now, to search for the names of two clerks we define the function *twoclerks* which is based upon $find$, which forces the desired pattern $e(N, S, clerk)$ over the set defined by $employees(branches)$. The resulting program is:

$$\begin{array}{cc} branches \to madrid & branches \to vigo \\ employees(madrid) \to e(pepe, men, clerk) & employees(madrid) \to e(paco, men, boss) \\ employees(vigo) \to e(maria, women, clerk) & employees(vigo) \to e(jaime, women, boss) \\ twoclerks \to find(employees(branches)) & find(e(N, S, clerk)) \to (N, N) \end{array}$$

With term rewriting $twoclerks \to find(employees(branches)) \not\to^* (pepe, maria)$, because in that expression the evaluation of *branches* is needed and so one of the branches must be chosen. On the other hand with $\pi CRWL$ (some steps have been omitted for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk)}\ \text{POR} \quad \dfrac{\dfrac{\dots}{employees(branches) \twoheadrightarrow e(maria, \bot, clerk)}\ \text{POR}}{\dots}\ \text{POR}}{(pepe\ ?\ maria, pepe\ ?\ maria) \twoheadrightarrow (pepe, maria)}\ \text{DC}}{find(employees(branches)) \twoheadrightarrow (pepe, maria)}\ \text{POR}}{twoclerks \twoheadrightarrow (pepe, maria)}\ \text{POR}$$

where

$$\dfrac{\dfrac{}{branches \twoheadrightarrow madrid}\ \text{POR} \quad \dfrac{\dots}{e(pepe, men, clerk) \twoheadrightarrow e(pepe, \bot, clerk)}\ \text{DC}}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk)}\ \text{POR}$$

We will also see some additional sample uses of $\pi CRWL$ in next section.

3.4.2 Implementing $\pi CRWL$ in Maude

In Sect. 3.2.2 we saw that neither *CRWL* can be simulated by term rewriting with a simple program transformation, nor vice versa. Nevertheless, plural semantics can be simulated by rewriting using the transformation presented here. We have used that transformation as one of the basis of a first implementation of $\pi CRWL$ [RR09a] that can be used for experimentation. These results first appeared in [Rod08] (Sect. 7.1.3, page 126) and [RR09b] (Sect. 7.2.3, page 140)

First we will present a naive version of this transformation, and show its adequacy; later we will propose some simple optimizations for it. Then we will briefly describe how we have exploited the Maude [CDE+07] metaprogramming capabilities to implement the transformation and the natural rewriting on-demand strategy [Esc04], which are the basis for this first prototype of $\pi CRWL$. Finally we will introduce the use of the system by means of some programs and a sample interpreter session.

The transformations

The first simple version of the transformation to simulate $\pi CRWL$with term rewrite is formulated as follows.

Definition 3.4.1 (Plural semantics transformation, simple version, [Rod08] Def. 13)**.** Given a *CRWL*-program P, for every rule $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ such that $f \notin \{\_?\_, if\_then\_\}$ we define its transformation as:

$$pST(f(p_1,\ldots,p_n) \to r) = f(Y_1,\ldots,Y_n) \to if\ match(Y_1,\ldots,Y_n)\ then\ r[\overline{X_{ij}/project_{ij}(Y_i)}]$$

- $\forall i \in \{1,\ldots,n\}$, $\{X_{i1},\ldots,X_{ik_i}\} = var(p_i) \cap var(r)$ and $Y_i \in \mathcal{V}$ is fresh.
- $match \in FS^n$ fresh is defined by the rule $match(p_1,\ldots,p_n) \to true$.
- Each $project_{ij} \in FS^1$ is a fresh symbol defined by the single rule $project_{ij}(p_i) \to X_{ij}$.

For $f \in \{\_?\_, if\_then\_\}$ the transformation leaves its rules untouched.

Notice that each rule R in $\mathcal{P}$ requires its own *match* and *project* functions. The function *match* is used to impose a "guard" that enforces the matching of each argument with its corresponding pattern. If we dropped this condition the translation of, for example, to rule $(null(nil) \to true)$, would be $(null(Y) \to true)$, which is clearly unsound as then $null(0 : nil) \to true$. Besides, each pattern p_i has been replaced by a fresh variable Y_i, to which any expression can match, hence the arguments may be replicated freely by the rewriting process without demanding any evaluation and thus keeping its denotation untouched: this is the key to achieve completeness wrt. $\pi CRWL$. Later on, the functions $project_{ij}$ will just make the projection of each variable when needed.

Example 3.4.3 ([Rod08] Ex. 14)**.** Applying this to Ex. 3.4.1 we get

$$f(Y) \to if\ match(Y)\ then\ d(project(Y),\\ project(Y)), match(c(X)) \to true, project(c(X)) \to X$$

under which we can do:

$$\underline{f(c(0)?c(1))} \to if\ \underline{match(c(0)?c(1))}\ then\ d(project(c(0)?c(1)), project(c(0)?c(1)))$$
$$\to^* \underline{if\ true\ then\ d(project(c(0)?c(1)), project(c(0)?c(1)))}$$
$$\to d(project(\underline{c(0)?c(1)}), project(\underline{c(0)?c(1)}))$$
$$\to^* d(\underline{project(c(0))}, \underline{project(c(1))}) \to^* d(0,1)$$

The following result summarizes our results concerning the adequacy of this transformation.

Corollary 3.4.2 ([Rod08] Corollary 17)**.** *For any CRWL-program $\mathcal{P}$, $e \in Exp$ built using symbols of the signature of $\mathcal{P}$, $[\![e]\!]^{pl}_{\mathcal{P}} = [\![e]\!]^{rw}_{pST(\mathcal{P})}$. Hence $\forall t \in CTerm\ \mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ iff $pST(\mathcal{P}) \vdash e \to^* t$.*

Concerning the transformation, if a pattern is ground then no parameter passing will be done for it and so no transformation is needed: for $null(nil) \to true$ we get $\{null(Y) \to if\ match(Y)\ then\ true,\ match(nil) \to true\}$, which is equivalent. Besides, if the pattern is a variable then any expression matches it and the projection functions are trivial, so no transformation is needed neither, as happens with $pair(X) \to (X,X)$ for which $\{pair(Y) \to if\ match(Y)\ then\ (project(Y),\ project(Y)), match(X) \to true, project(X) \to X\}$ are returned.

Definition 3.4.2 (Plural semantics transformation, optimized version, [Rod08] Def. 18)**.** Given a *CRWL*-program P, for every rule $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ we define its transformation as:

$$pST(f(p_1,\ldots,p_n) \to r)$$
$$= \begin{cases} f(p_1,\ldots,p_n) \to r & \text{if } \rho_1 \ldots \rho_m \text{ is empty} \\ f(\tau(p_1),\ldots,\tau(p_n)) \to \begin{array}{l} if\ match(Y_1,\ldots,Y_m) \\ \quad then\ r[\overline{X_{ij}/project_{ij}(Y_i)}] \end{array} & \text{otherwise} \end{cases}$$

where $\rho_1 \ldots \rho_m = p_1 \ldots p_n \mid \lambda p.(p \notin \mathcal{V} \wedge var(p) \neq \emptyset)$.
- $\forall \rho_i$, $\{X_{i1},\ldots,X_{ik_i}\} = var(\rho_i) \cap var(r)$ and $Y_i \in \mathcal{V}$ is fresh.
- $\tau : CTerm \to CTerm$ is defined by $\tau(p) = p$ if $p \in \mathcal{V} \vee var(p) = \emptyset$ and $\tau(p) = Y_i$ otherwise, for $p \equiv \rho_i$.
- $match \in FS^m$ fresh is defined by the rule $match(\rho_1,\ldots,\rho_m) \to true$.
- Each $project_{ij} \in FS^1$ is a fresh symbol defined by the single rule $project_{ij}(\rho_i) \to X_{ij}$.

We will not give a formal proof for the adequacy of the optimization. Nevertheless note how this transformation leaves untouched the rules for $\_?\_$ and $if\_then\_$ without defining a special case for them. As the simple transformation worked well for that rules that suggests that we are doing the right thing. In the following example we apply the optimized transformation to the program of Ex. 3.4.2.

Example 3.4.4 ([Rod08] Ex. 19)**.** The only rule modified is the one for $find$, for which we get $\{find(Y) \to if\ match(Y)\ then\ (project(Y), project(Y)), match(e(N,s,clerk)) \to true, project(e(N,s,clerk)) \to N\}$, so:

$$\underline{twoclerks} \to \underline{find(employees(branches))}$$
$$\to if\ match(\underline{employees(branches)}) \\ \quad then\ (project(employees(branches)), project(employees(branches)))$$
$$\to^* \underline{if\ match(e(pepe,men,clerk))} \\ \quad \underline{then\ (project(employees(branches)), project(employees(branches)))}$$
$$\to^* (project(\underline{employees(branches)}), project(\underline{employees(branches)}))$$
$$\to^* (\underline{project(e(pepe,men,clerk))}, \underline{project(e(maria,women,clerk))} \to^* (pepe, maria)$$

Implementing $\pi CRWL$ in Maude

Maude [CDE+07] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in Meseguer's *rewriting logic* [MM02], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems.

This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [BJM00], which, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic also extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude *system modules* are used to define specifications in this logic, hence in particular Maude can be used as a rewriting machine for possible non-confluent and non-terminating CS's, which are the kind of programs generated by the transformation that simulates $\pi CRWL$.

Exploiting the fact that rewriting logic is reflective [CMP07], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [CDE+07, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as Maude modules, expressions or computations as usual data. In addition, the Maude system provides another module, `LOOP-MODE` [CDE+07, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our program transformation, its execution, and its user interactions are implemented in Maude itself.

Although Maude provides commands to execute expressions in (metarepresented) modules, including a `metaSearch` function that performs a breadth-first search of the state space,[7] the highly non-deterministic nature of the programs obtained with the transformation avoids its use in practice. To solve this problem we have implemented the natural rewriting strategy [Esc04], that evolves only the terms needed in the execution of an expression, avoiding to rewrite unnecessary terms. This is the first implementation of an on-demand strategy for Maude system modules,[8] and it can be considered a first stage towards on-demand execution of general rewrite theories.

This on-demand strategy has been combined with depth-first and breadth-first search, which allows to traverse the search tree in a flexible way, allowing to evaluate programs with potentially infinite branches. Furthermore, the tool also provides the option of searching with a bound in the number of rewrites, thus enhancing the performance of programs with large (possibly infinite) error branches.

Implementing the transformation in Maude As we have already mentioned, the reflection capabilities of Maude allow us to manipulate (metarepresented) Maude modules (and, more concretely, Maude rules) as data. Hence we have implemented the optimized transformation $pST()$ as a function `pST` that receives the rule that must be transformed and an index to create fresh function names related to this rule and returns a set of rules composed by the new rule and the associated *match* and *project* rules. This corresponds to the following Maude operator declaration: `op pST : Rule Nat -> RuleSet`. More details about the concrete Maude code can be found in [RR09b] (Sect. 7.2.3, page 140) and in the source code of the system [RR09a].

Implementing natural rewriting in Maude The second component of our system is an implementation of the natural rewriting on-demand strategy [Esc04], which became necessary to deal with the highly non-deterministic programs obtained after the transformation. As it is usual in other on-demand strategies, a data structure called *definitional tree* [Ant92] is used to encode the demand information associated to the program rules. What makes natural rewriting different and allows it to perform a better treatment of demandness than other on-demand strategies, is that it uses a special kind of definitional tree called *matching definitional tree*, that allows us to keep the pattern matching process separated from the evaluation through demanded positions. In previous strategies the encoding of these two processes were interleaved in the definitional trees, and as a consequence they lost opportunities to prune the search space. A matching definitional tree

[7] The usual Maude strategy, consisting in rewriting terms with the first possible rule is not appropriate here because it leads to results that are not necessarily c-terms.

[8] On-demand strategies for Maude functional modules are described in [DEL05].

is built for each function present in the program, after a static analysis performed during the compilation. We implement this by the operator `MDTMap`, which takes the Maude representation of the transformed program and returns a map from function symbols to its corresponding definitional trees.

Once the matching definitional trees have been computed, we can use the function `mt` [Esc04] to compute the needed positions and the rules that must be applied. Moreover, we combine this evaluation strategy with (bounded) depth-first and breadth-first search, keeping *all* the possible terms obtained from `mt` and its depth in a list that works as a stack for the depth-first strategy and as a queue for the breadth-first strategy. Finally the main function that implements the strategy is the operator `natNext`, that given a term, a program and the map generated by `MDTMap`, computes the set of terms reachable from the given term by applying the strategy.

Using the prototype

Now we will illustrate the use of the tool [RR09a] by means of some examples. First, we specify the clerks program from Ex. 3.4.2 (page 97) in Sect. 3.4.1:

```
Maude> (plural CLERKS is
  branches -> madrid .
  branches -> vigo .
  employees(madrid) -> e(john, men, clerk) .
  employees(madrid) -> e(larry, men, boss) .
  employees(vigo) -> e(mary, women, clerk) .
  employees(vigo) -> e(james, men, boss) .
  twoclerks -> find(employees(branches)) .
  find(e(N,S,clerk)) -> p(N,N) .
endp)

Module introduced.
```

Under $\pi CRWL$ the expression `twoclerks` leads to any combination `p(name`$_1$`, name`$_2$`)`, where `name`$_i$ can be any clerk name (`john` and `mary` in the example), while run-time choice and call-time choice only lead to pairs where `name`$_1$ and `name`$_2$ coincide.

The tool reads the module and applies it the `pST` transformation, that simulates plural semantics with ordinary rewriting. This transformed module can be seen with the command `(showTr .)`. We can now change the default depth-first strategy to breadth-first by using the command

```
Maude> (breadth-first .)

Breadth-first strategy selected.
```

We try now to compute the result of evaluating `twoclerks` by typing

```
Maude> (eval twoclerks .)

Result: p(john,john)
```

Since we try to find results with different names in the pair, we can ask for more answers with

```
Maude> (more .)

Result: p(john,mary)
```

The `more` command allows us to perform backtracking, so we can repeatedly use it to obtain all the different pairs reachable by the program until the following answer is prompted:

```
Maude> (more .)

No more results.
```

Now that we are familiar with the tool we show how to execute a more complex problem. The fearless Ulysses has been captured in his travel from Troy to Ithaca, but he knows he can persuade one of his four guardians to interchange the key for some items, that Ulysses has to obtain from the other guardians with his initial possessions:

```
(plural LAIR is
  guardians -> circe ? calypso ? aeolus ? polyphemus .
  ask(circe, trojan-gold) -> item(treasure-map) ? sirens-secret .
  ask(calypso, sirens-secret) -> item(chest-code) .
  ask(aeolus, item(M)) -> combine(M,M) .
  ask(polyphemus, combine(treasure-map, chest-code)) -> key .
```

Notice that the information given to the fourth guardian can be only obtained with our semantics, because a pair of the same variable becomes a pair of different constants. To acquire these items he uses the function `discover`, that uses the current information or tries to ask the guardians for more.

```
  discover(M) -> M ? discover(discStep(M) ? M) .
  discStep(M) -> ask(guardians, M) .
```

Finally, Ulysses escapes if he obtains the key from his initial belongings: an immeasurable amount of `trojan-gold`.

```
  escape -> open(discover(trojan-gold)) .
  open(key) -> true .
endp)
```

We use the depth-first strategy to check if the evasion is possible:

```
Maude> (depth-first .)

Depth-first strategy selected.
```

We evaluate now the term `escape` with `80` as upper bound in the number of rewrites with the command:

```
Maude> (eval [depth= 80] escape .)

Result: true
```

That is, there is a way to interchange the information in order to escape.

3.4.3 A Plural Semantics for Constructor Systems: conclusions

In this section we have gone through our results about the novel semantics $\pi CRWL$, first presented in [Rod08] (Sect. 7.1.3, page 126) and [RR09b] (Sect. 7.2.3, page 140). This semantics came up naturally through our searching for a new rewriting logic for term rewriting, and here we have pointed the different interpretations of run-time choice and plural semantics caused by pattern matching. To the best of our knowledge this distinction was established in [Rod08] for the first time, because in [SS92] no pattern matching was present and in [Hus93] only call-time choice was adopted. We argue that the run-time choice semantics induced by term rewriting is not the best option for a value-based programming language like current implementations of FLP. For that context a plural semantics has been proposed for which the compositionality properties lost when turning from call-time choice to rewriting are recovered. Nevertheless, for other kind of rewriting based languages like Maude, which are not limited to constructor-based TRS's, term rewriting has been proven to be an effective formalism.

We have not only formalized the new semantics through the $\pi CRWL$ proof calculus, but also have proved some of its fundamental properties, and show how it allows natural encodings of some programs that need to do some collecting work. Then we have compared the new calculus with $CRWL$ and term rewriting, proving the inclusion chain $CRWL \subseteq$ rewriting $\subseteq \pi CRWL$ wrt. the set of computed c-terms. Finally, although it is impossible to get a straight simulation of $CRWL$ in term rewriting nor vice versa—as seen in Sect. 3.2.2—, we have proved that it is possible to simulate our plural semantics with term rewriting, by providing a simple program transformation. This transformation has been implemented in Maude [CDE+07], and together with an implementation of the natural rewriting on-demand strategy [Esc04], it constitutes the basis for our first prototype of $\pi CRWL$ [RR09a].

On the other hand, that implementation of the natural rewriting strategy is an important side contribution of our work in [RR09b]. The corresponding `natNext` operator can be used for performing on-demand evaluation of any CS specified in a Maude system module, and it is especially relevant because it is the first on-demand strategy for this kind of modules, complementing the default rewrite and breadth-first search Maude commands.

From a practical point of view, it might be unrealistic to think that a monolithic semantic view is adequate for addressing all non-determinism present in a large program. In Sect. 3.3.3 we saw our proposals for the combination of call-time choice and run-time choice in a unified framework. But as $\pi CRWL$ seems to be more suitable than run-time choice for a value-based language, we are extending that work to $\pi CRWL$. We have already obtained interesting results in that direction: in [RR10] we can find an extension of $\pi CRWL$ to allow the combination of plural and singular arguments. This new features have been already added to our prototype [RR09a], and are very useful for developing programs to explore the expressive capabilities of our plural semantics.

We also contemplate other relevant subjects of future work, like extending the theory to handle programs with extra variables; studying the possible equivalence of call-time choice, run-time choice and $\pi CRWL$ for deterministic programs; developing an operational notion for $\pi CRWL$ at the abstraction level of *let*-rewriting; devising the corresponding narrowing notion; or extending the semantics with new features like higher order functions or matching-modulo capacities—a characteristic of Maude—, and adding them to our

prototype.

Chapter 4

Conclusions and Future Work

"HELENA: Good heavens, they have to work immediately?
DOMIN: Sorry. They work the same way new furniture works. They get broken in. Somehow they heal up internally or something. Even a lot that's new grows up inside them. You understand, we have to leave a bit of room for natural development. And in the meantime the products are refined.
HELENA: How do you mean?
DOMIN: Well, it's the same as "school" for people. They learn to speak, write, and do calculations. They have a phenomenal memory. It you were to read them a twenty-volume encyclopedia they could repeat the contents in order, but they never think up anything original. They'd make fine university professors. Next they are sorted by grade and distributed. Fifty thousands head a day, not counting the inevitable percentage of defective ones that are thrown into the stamping-mill . . . etcetera, etcetera."

Karel Capek — R.U.R. (Rossum's Universal Robots) - 1921

4.1 Contributions

In this work we have tried to make some contributions to the field of non-deterministic functional-logic programming. Let us summarize these, to see how close have we got to achieving our original goals.

1. **Providing new descriptions of existing semantics for non-determinism in TRS's**.

   a) *For call-time choice*: emphasis on *rewriting-like operational models*, as *CRWL* already provides a declarative semantics for call-time choice.

   We have developed a modification of term rewriting called *let*-rewriting [LRS07b], inspired in [AFM+95, MOW98, Plu99, SH04], in which the notions of subexpressions sharing and call-time choice are added to the framework of term rewriting. The key idea is extending the syntax with a *let* construction and then defining a reduction relation which manages these *lets* properly, and forbids parameter passing for unshared function arguments.
   We have also presented its associated *let*-narrowing relation [LRS09d] and extended both notions to higher order too, thus getting the *HOlet*-rewriting and

HOlet-narrowing relations [LRS08a]. We have proved the adequacy of *let*-rewriting and *HOlet*-rewriting wrt. the consolidated frameworks of *CRWL* and *HOCRWL* respectively, thus showing that these reduction systems describe a call-time choice semantics as it is implemented in current FLP systems. The adequacy of each narrowing relation wrt. to its corresponding rewriting relation has been proved too, being the key a lifting lemma in the style of Hullot's [Hul80]. Finally we have outlined some applications of the higher order version of the framework, namely the proof of the correctness of bubbling [ABC07] and of the higher order to first order translation [War82] used in mainstream FLP implementations.

b) *For run-time choice*: emphasis in *declarative semantics*, as term rewriting already provides a simple notion of reduction step for run-time choice.

We have proposed a new rewriting logic for CS's [LRS09b] that is based on a structured representation of sets of c-terms, called $SCTerm$'s. This logic enjoys some nice properties: polarity, monotonicity of substitutions, closedness under substitutions and, above all, compositionality wrt. $SCTerm$'s and full abstraction wrt. sensible notions of 'observable'. As *CRWL* enjoys very similar properties we could then adapt the reasoning techniques based on *CRWL* to the framework of CS's through our new logic. For instance, we could use an approach based on information orders like the approximation order $\sqsubseteq$ instead of the more classical techniques based on derivation reconstruction and descendants tracing. And we can do it because of the strong adequacy results obtained for our logic wrt. term rewriting. Moreover the domination relation $\_ \lessdot \_$ conceived during those adequacy proofs is an interesting side product whose usefulness for reasoning about CS's should be studied in the near future.

Regarding the related side goals of our first main goal:

- *Connecting several semantic descriptions of modern functional-logic programming*:

  In [LRS07a] we made a precise technical comparison between the semantics described in the *CRWL* framework and in the operational semantics FLC of [AHH+05]. There we obtained equivalence results for a wide class of programs and expressions, thereby getting the first results establishing a real technical connection between these important frameworks in the field of FLP.

  On the other hand, in [LRS07b, LRS09d] we made a comparison between call-time choice and run-time choice regarding the set of computed c-terms, showing that call-time choice computes strictly less values in general and exactly the same values for deterministic programs, and extended it to their narrowing versions.

- *Studying the full abstraction problem for non-deterministic rewriting-based languages*: We have contributed to this goal in two different directions:

  – For *run-time choice*: In [LRS09b] we proved the full abstraction of our new semantics wrt. natural observation notions based on term rewriting, like the set of c-terms reachable by rewriting, or the set of outer constructed part

of the terms reachable by rewriting.

  - For *call-time choice*: In [LR10] the problem was studied in a higher order setting, using *HOCRWL* as the semantic notion and *HOlet*-rewriting as the operational one, so the considered observations were the set of patterns or first order patterns reachable by *HOlet*-rewriting. Positive results were obtained for the class of programs without extra variables.

- *Experimenting with the use of automatic theorem provers or proof assistants for reasoning about the semantics of functional-logic programming*: In [LMR09a] a formalization of *CRWL* in the Isabelle proof assistant was presented, in which some fundamental properties of *CRWL* like polarity, closedness under c-substitutions and compositionality for c-terms are formally proved in the system.
  The corresponding Isabelle library can be found in [LMR09c].

2. **Investigating new semantics alternatives**.

   a) Semantic combinations: We have proposed two different possible combinations of call-time choice and run-time choice in the same language.

      - In a *run-time choice environment*: In [LRS09a] we start from the same syntax used in *let*-rewriting but now allowing to perform parameter passing for unshared function arguments. The resulting relation, called *rt-let*-rewriting, performs run-time choice by default but through the *let* construction it also allows to specify that the values of some expression will be shared, thus getting a conservative extension of both call-time choice and run-time choice, as it is formally proved in the paper.
        A prototype implementation based on the *Toy* system was developed as a result [LRS09e].

      - In a *call-time choice environment*: In [LRS09c] we go in the opposite direction, we start from a regular call-time choice framework in which everything is shared, and then we extend it by providing primitives to "unshare" particular expressions.
        Again a prototype implementation was developed as an extension of *Toy* [LRS08b].

   b) A plural semantics with pattern matching: In [Rod08] besides formalizing the new semantics through a modification of *CRWL* called $\pi CRWL$, we proved the polarity and compositionality of this new semantics, and several monotonicity properties of substitutions under it. Then we extended the semantic hierarchy that we started with our comparison between call-time choice and run-time choice, concluding that the novel plural semantics is strictly bigger than run-time choice, and as a consequence, than call-time choice.
      Regarding an implementation of this new semantics, in [RR09b] we used the program transformation proposed in [Rod08] to implement a first prototype interpreter for $\pi CRWL$ [RR09a] in the Maude system. The soundness of this implementation is based on the adequacy of the simulation performed by the transformation, shown in [Rod08]. Several examples trying to show the interest of the new semantics can be found in both works.

When developing this prototype we have also obtained an important side contribution, by implementing the natural rewriting strategy for Maude system modules. This is especially relevant because it is the first on-demand strategy for this kind of modules, complementing the default rewrite and breadth-first search Maude commands.

Taking a look at these results we can conclude that we have made significant advances in each of our original goals. Nevertheless, as those objectives were pretty general, there is still a lot of work to be done in those lines. We will outline some possible extensions of our work in the next section.

4.2 Future Work

The following are some possible lines opened for future work.

- *On-demand strategies for let-rewriting and let-narrowing*: These reduction notions still cannot be considered to be effective operational notions at the practical level, because of the absence of a redex selection strategy that could determine for each step which rule to apply and over which subexpression. There are several on-demand strategies for term rewriting and narrowing, being needed narrowing [AEH94] and natural narrowing [EMT05] two of the most well known in the field of FLP. In practice the use of some strategy is indispensable because otherwise the search space of the computation might grow until it becomes unmanageable.

  Our reduction notions were conceived on one hand to be independent of a particular on-demand strategy, and on the other hand to ease the adoption of any concrete strategy. Therefore we consider that the lack of an implicit strategy is an advantage more than a defect. In some future work we could investigate the use of strategies to develop relations derived from *let*-rewriting and *let*-narrowing, that would be subrelations of those, as the strategies are used to reduce the search space by avoiding unnecessary steps. We think that *let*-rewriting could be just the right tool to prove the optimality and adequacy of those strategies when used in a call-time choice environment, which is still an unresolved problem, as those strategies were originally formulated for term rewriting, that is, for run-time choice.

- *Extensions of let-rewriting*: We have already extended the framework of *let*-rewriting to the higher order setting established by *HOCRWL* [GHR97], so we think that the following natural step would be using it to solve some well known problems about type systems that have arisen in FLP. There have been already some advances in that line, in particular in [LMR10] we used *HOlet*-rewriting and an extension of Damas & Milner type system [DM82] to fix a violation of the subject reduction property related to higher order patterns.

  Another challenging problem, already mentioned in [LRS08a] and known from afar [GHR01], is the violations of the subject reduction property caused by narrowing with higher order variables, as well as the consequent uncontrolled growth of the search space. In [GHR01] a goal solving calculus enhanced with type information was proposed to solve this problem, maybe those ideas could be adapted to *HOlet*-narrowing too.

Finally we could extend *let*-rewriting and narrowing to handle constraints, which are also a common feature of FLP systems.

- *Advances in our rewriting logic for CS's*: The logic developed in [LRS09b] was devised with CS's in mind, and so it uses the notion of constructor to define its denotational domain. The constructor discipline is a well-accepted programming discipline which is adopted in most functional languages, and in fact equational programs that violate the constructor discipline are rare in practice [O'D85]. Nevertheless, as a consequence of using a domain based on the notion of constructor, our logic cannot be used in other kinds of TRS's which do not follow the constructor discipline. Therefore it would be interesting trying to overcome this limitations, by replacing the role of constructor values by appropriate alternatives. We think that a promising approach would be to take the ideas of [Tha85, DS93, Sal95], where it is proved that any orthogonal TRS can be transformed into an equivalent CS, and adapt them to define a new denotational domain that we could then use to define an extension of our logic for a more general class of TRS's.

- *Comparison of semantic descriptions of call-time choice*: Until now our comparison only covers *CRWL*, term rewriting and the operational semantics FLC of [AHH+05]. Furthermore, the equivalence results between *CRWL* and FLC are given for a restricted class of programs. First of all we would like to extend these results to cover the full class of *CRWL* programs. Maybe our *let*-rewriting relation could be a suitable tool for that task, as it is an operational notion and so it is much closer to *FLC* than *CRWL*. On the other hand we would find interesting proving the expected equivalence of *CRWL* to the formalization of graph rewriting of [EJ97, EJ98], that is also an important family of semantic descriptions that have been used in many works about FLP, for example [ABC07, ABC06].

- *Comparison of semantics for non-determinism under deterministic programs*: In Sect. 3.3.2 we proved the equivalence of *CRWL* and term rewriting for the class of deterministic programs, and strongly conjectured that the confluence of a *CRWL*-program under term rewriting implies its determinism. First of all note that determinism was defined there in terms of *CRWL*: a program is deterministic whenever for any expression its denotation *under CRWL* is a directed set. Thus we could imagine two other notions of determinism of programs by using the denotations defined either by the semantics for term rewriting of Sect. 3.3.1, or by the $\pi CRWL$ logic of Sect. 3.4.1. Proving the former conjecture, studying the relations between these later notions of determinism, and finally investigating the possible equivalence of *CRWL*, term rewriting and $\pi CRWL$ under different assumptions of determinism, are some open lines of future work on this subject.

- *The full abstraction problem*: As already discussed in Sect. 3.2.5, for higher order languages under call-time choice semantics we only got positive results for programs without extra variables, hence it would be interesting to overcome this limitations. We think this could be possible by extending the syntax with lambda abstractions, which is an old pending issue for the *CRWL* framework. Although a variation of *CRWL* to support lambda abstractions is proposed in [Vad07, Vad09], it follows a different approach to the logic *HOCRWL* presented in Sect. 3.2.4 and originally developed in [GHR97, GHR01], because the notions of program and value used

there are fundamentally different to those used in *HOCRWL*. The logic proposed in [Vad07, Vad09] describes the semantics of a language essentially different to mainstream FLP systems like *Toy* or *Curry*, while *HOCRWL* provides a description for a subset of these languages, in particular not including support for lambda abstractions. The addition of lambda abstractions to these languages is a pending task for the whole FLP community too, because although some FLP systems have support for lambda abstractions [Lux07, Han09], these have different behaviours in different systems. Henceforth an exploration of the design space for the introduction of lambda abstraction in systems like *Toy* or *Curry*, by giving formal characterizations for the different alternatives and investigating their properties, would be interesting in any case.

Another orthogonal extension of our results regarding full abstraction could go in the line of giving a more active role to variables, that until now have been treated almost like constants in our works. This could be interesting taking into account that variables can be instantiated by narrowing, or simply by the instantiation made in parameter passing by rewriting. We think that this extension could be used as a powerful tool for program transformation, giving an important necessary condition for the replacement of right hand sides of program rules.

- *Mechanized theorem proving*: As pointed out before, we have just given the first steps in this direction. One possible extension of our work would be trying to combine our results about the metatheory with previous works about *CRWL* and Isabelle [CLLF04], which put the focus on proving properties of concrete programs.

  Another possible subject of future work in this line would be formalizing the *let*-rewriting relation in Isabelle and then proving its adequacy wrt. *CRWL* in this system. This would be a step in the direction of challenge 3 of [ABF+05], "Testing and Animating wrt. the Semantics", because we would end up getting an interpreter of *CRWL* during the process. We should then also formalize some on-demand evaluation strategy for our Isabelle formalization of *let*-rewriting, thus obtaining an Isabelle proof of its optimality.

  We could do something similar with our semantics for constructor systems of Sect. 3.3.1. Its Isabelle formalization should we similar to the formalization of *CRWL*, and we think the implementation in Isabelle of its associated operational notion, that is, term rewriting, would be very easy as well, as it is a pretty much simpler notion that *let*-rewriting, and whose formalization in Isabelle has been already tackled in the IsaFoR (Isabelle Formalization of Rewriting) library which is part of the CeTA (Certified Termination Analysis) system [TS09].

- *Semantic combinations*: In [LRS09a] we already pointed out a possible application of our combination of call-time choice and run-time choice in a run-time choice environment. The point is that most FLP systems implement sharing by means of an internal construction called suspension, that plays a role parallel to the *let* bindings of *rt-let*-rewriting. Therefore we think that this could be the most suitable abstraction level for reasoning about suspensions in order to minimize its creation, thus increasing the performance of systems.

  On the other hand the combination of call-time choice and the plural semantics expressed by $\pi CRWL$ deserves to be investigated, as both semantics are compositional

wrt. c-terms, thus allowing a value-based programming style. We will also deal with this matter in the next item.

- *Plural semantics*: There is a lot of work to be done about $\pi CRWL$. First of all its expressive capabilities should be studied in depth in order to get more interesting programming patterns that could exploit the capabilities of this new semantics. We have already advanced in that direction in [RR10]. There becomes apparent that the combination of plural and singular arguments in the same language makes writing programs easier than in a monolithic plural semantics, and at the same time that the use of plural arguments in some fragments of the programs arises naturally and helps to improve the declarative flavour of programs.

  Then an operational notion at the level of *let*-rewriting should be devised for $\pi CRWL$, as the current transformational implementation performs a lot of duplication of computations. Some sharing of sets of computations in the line of [BH07] should be performed in that reduction notion. Another interesting goal would be designing a new narrowing procedure that would be complete for $\pi CRWL$, as current narrowing relations are complete only for normalizing substitutions, a condition not fulfilled by the substitutions used in $\pi CRWL$ for parameter passing. Other possible extensions of the framework could include the support for extra variables in $\pi CRWL$ and the addition of higher order features or even matching-modulo capacities—a prominent trait of the rewriting logic of Meseguer [MM02], which is implemented in Maude [CDE+07].

  Finally the notion of $SCTerm$ of [LRS09b] seems to be a more natural value for $\pi CRWL$. Although we were able to formulate it by using c-terms only, some difficult to express properties like the closedness of substitutions can be naturally treated by using $SCTerm$'s. This suggests a possible unified semantic framework in which the semantics of call-time choice, run-time choice and $\pi CRWL$, and any combination of them, could be expressed by means of a single logic.

- *Semantic alternatives in non-deterministic lambda calculus*: In [KSS98] a non-deterministic extension of the lambda calculus was presented. This framework was extended in [SSH00] with constructors and *letrec* and *case* primitives. This gives us the ingredients needed to translate the semantic hierarchy established for non-deterministic term rewriting systems to the world of non-deterministic lambda calculus. We find this line quite appealing also because the techniques used in those works are fundamentally different, and are based in the notions of observational equivalence and bisimulation. Moreover, in those works there is a concern in the demonic/angelic/erratic dimension of non-determinism, something that has been taking apart in our works, focused only in angelic non-determinism. Therefore we think that trying to transfer ideas between these two research lines can be fruitful for both of them.

- *Algorithmic aspects of non-determinism*: As we mentioned in the introduction, several works in the field of functional programming have been developed to simulate it by different constructions [Wad85, Hin00, KSFS05, FBK05, NAR07, FKS09]. We also made a small contribution to this subject in [LRS07c]. We would find interesting to deepen in this line, trying to obtain a more systematic procedure for defining data structures for representing non-determinism, but also trying to represent not only the

usual call-time choice semantics represented in previous works, but also the different semantics presented in this thesis, and their possible combinations.

Chapter 5

Conclusiones y trabajo futuro

"HELENA: Dios mío, ¿tienen que ponerse a trabajar inmediatamente?
DOMIN: Claro, mire, funcionan de la misma forma que el mobiliario nuevo. Se van amoldando. De alguna manera se asientan por dentro o algo similar. Incluso crecen bastantes cosas nuevas en su interior. Ya sabe, tenemos que dejar un poco de espacio para que las cosas se desarrollen de forma natural. Y mientras tanto los productos se van refinando.
HELENA: ¿Que quiere decir?
DOMIN: Bueno, es lo mismo que el "colegio" para las personas. Aprenden a hablar, a escribir y a hacer cuentas. Tienen una memoria fenomenal. Si les leyera una enciclopedia de veinte volúmenes serían capaces de repetir sus contenidos en orden, pero sin embargo no son capaces de concebir nada nuevo. Serían buenos profesores de universidad. Después son ordenados por categorías y distribuidos. Cincuenta mil cabezas al día, sin contar el inevitable porcentaje de unidades defectuosas que se tiran a la trituradora de basura . . . etcétera, etcétera."

Karel Capek — R.U.R. (Robots Universales Rossum) - 1921

5.1 Contribuciones

En este trabajo hemos intentado hacer algunas contribuciones al campo de la programación lógico-funcional indeterminista. Procederemos ahora a resumirlas, para ver hasta que punto nos hemos acercado a la consecución de nuestros objetivos originales.

1. **Proporcionar nuevas descripciones para semánticas existentes del indeterminismo en TRS's**.

   a) *Para call-time choice*: énfasis en los *modelos operacionales al estilo de la reescritura*, ya que *CRWL* ya proporciona una semántica declarativa para call-time choice.

      Hemos desarrollado una modificación de la reescritura de términos llamada *let*-reescritura [LRS07b], inspirada en [AFM+95, MOW98, Plu99, SH04], en la que las nociones de compartición de subexpresiones y call-time choice han sido añadidas al marco de la reescritura de términos. La idea clave es extender la sintaxis con una construcción *let* y definir entonces una relación de reducción que maneje

esos *lets* de manera apropiada, y a la vez prohíba realizar el paso de parámetros para argumentos de función que no estén compartidos.

También hemos presentado su relación de *let*-estrechamiento asociada [LRS09d] y extendido ambas nociones para soportar orden superior, de esta manera obteniendo las relaciones de *HOlet*-reescritura y *HOlet*-estrechamiento [LRS08a]. Hemos demostrado la adecuación de la *let*-reescritura y la *HOlet*-reescritura con respecto a los marcos consolidados de *CRWL* y *HOCRWL* respectivamente, mostrando por tanto que estos sistemas de reducción describen una semántica de call-time choice tal y como es implementada en los sistemas FLP actuales. La adecuación de cada relación de estrechamiento con respeto a su relación de reescritura correspondiente ha sido demostrada también, siendo el elemento clave de estas demostraciones un lema de elevación al estilo del de Hullot [Hul80]. Para concluir hemos repasado algunas posibles aplicaciones de la versión de orden superior de este marco, en concreto la prueba de la corrección del bubbling [ABC07] y la transformación de orden superior a primer orden [War82] usada en la mayoría de las implementaciones de FLP.

b) *Para run-time choice*: énfasis en la *semántica declarativa*, ya que la reescritura ya proporciona una noción simple de paso de reducción para run-time choice.

Hemos propuesto una nueva lógica de reescritura para CS's [LRS09b] que está basad en una representación estructura de conjuntos de c-términos, llamados $SCTerm$'s. Esta lógica disfruta de buenas propiedades: polaridad, monotonía de las sustituciones, cierre bajo sustituciones y, sobre todo, composicionalidad con respecto a los $SCTerm$'s y full abstraction con respecto a nociones razonables de observable. Como *CRWL* también disfruta de propiedades similares, entonces podríamos adaptar las técnicas de razonamiento basadas en *CRWL* al marco de los CS's por medio de nuestra lógica. Por ejemplo podríamos usar un enfoque basado en órdenes de información como el orden de aproximación $\sqsubseteq$ en vez de técnicas más clásicas basadas en reconstrucción de derivaciones y traza de descendientes. Y podemos hacer esto gracias a los resultados fuertes de adecuación obtenidos para nuestra lógica con respecto a reescritura de términos. Además la relación de dominación $\_ \lessdot \_$ concebida durante dichas demostraciones de adecuación es un resultado secundario interesante cuyos utilidad para el razonamiento sobre CS's debería ser estudiada en un futuro próximo.

Respecto a los objetivos secundarios relacionados con nuestro primer objetivo principal:

- *Conectar varias descripciones semánticas de la programación lógico-funcional moderna*:

  En [LRS07a] realizamos una comparación técnica precisa entre la semántica descrita por el marco *CRWL* y aquella descrita en la semántica operacional FLC de [AHH+05]. En ese trabajo obtuvimos resultados de equivalencia para una amplia clase de programas y expresiones, de esta manera logrando los primeros resultados que establecen una relación técnica real entre estos importantes marcos en el campo de la FLP.

Por otra parte, en [LRS07b, LRS09d] realizamos una comparación entre call-time choice y run-time choice respecto al conjunto de c-términos calculados, mostrando que call-time choice calcula estrictamente menos valores en general y exactamente los mismos para programas deterministas, y además extendimos esta comparación a sus versiones de estrechamiento.

- *Estudiar el problema de la full abstraction para lenguajes indeterministas basados en reescritura*: Hemos contribuido a este objetivo en dos direcciones diferentes:
  - Para *run-time choice*: En [LRS09b] probamos la full abstraction de nuestra semántica respecto a nociones de observación naturales basadas en reescritura de términos, como el conjunto de c-términos alcanzables mediante reescritura, o el conjunto formado por las partes construidas externas de los términos alcanzables mediante reescritura.

  - Para *call-time choice*: En [LR10] este problema fue estudiado en un montaje de orden superior, usando *HOCRWL* como la noción semántica y *HOlet*-reescritura como la noción operacional, por lo que las observaciones consideradas fueron los conjuntos de patrones o de patrones de primer orden alcanzables mediante *HOlet*-reescritura. Se obtuvieron resultados positivos para la clase de programas sin variables extra.
- *Experimentar con el uso de demostradores de teoremas o asistentes de demostración para el razonamiento acerca de las semántica de la programación lógico-funcional*: En [LMR09a] presentamos una formalización de *CRWL* en el asistente a la demostración Isabelle, en la que algunas propiedades fundamentales de *CRWL* como la polaridad, el cierre bajo c-sustituciones y la composicionalidad para c-términos se probaron formalmente en dicho sistema.
  La biblioteca de Isabelle correspondiente puede encontrarse en [LMR09c].

2. **Investigar acerca de nuevas alternativas semánticas**

   a) Combinación de semánticas: Hemos propuesto dos posibles combinaciones de call-time choice y run-time choice en el mismo lenguaje.

   - En un *entorno con run-time choice*: En [LRS09a] partimos de la misma sintaxis utilizada para la *let*-reescritura pero en este caso permitiendo el paso de parámetros para argumentos de función sin compartir. La relación resultante, llamada *rt-let*-reescritura, realiza run-time choice por defecto pero a través de la construcción *let* también permite especificar que los valores de una expresión serán compartidos, obteniendo de esta manera una extensión conservadora de tanto call-time choice como run-time choice, como se demuestra formalmente en el artículo.
     Una implementación prototípica basada en el sistema *Toy* fue desarrollada a consecuencia [LRS09e].

   - En un *entorno con call-time choice*: En [LRS09c] vamos en la dirección opuesta, empezando por el entorno usual de call-time choice en FLP en el que todo está compartido, que extendemos proporcionando primitivas para "descompartir" expresiones particulares.

De nuevo una implementación prototípica fue desarrollada como una extensión de *Toy* [LRS08b].

b) *Un semántica plural con ajuste de patrones*: En [Rod08] además de formalizar la nueva semántica a través de una modificación de *CRWL* llamada $\pi CRWL$, demostramos la polaridad y composicionalidad de esta nueva semántica, y varias propiedades de monotonía para las sustituciones bajo esta. Entonces extendimos la jerarquía semántica que empezamos con nuestra comparación entre call-time choice y run-time choice, concluyendo que la nueva semántica plural es estrictamente más grande que run-time choice, y como consecuencia, que call-time choice.

En cuanto a la implementación de esta nueva semántica, en [RR09b] empleamos la transformación de programa propuesta en [Rod08] para implementar un primer intérprete prototípico para $\pi CRWL$ [RR09a] en el sistema Maude. La corrección de esta implementación está basada en la adecuación de la simulación realizada por la transformación, mostrada en [Rod08]. Varios ejemplos tratando de mostrar el interés de la nueva semántica pueden encontrarse en ambos trabajos.

Durante el desarrollo de dicho prototipo también obtuvimos una importante contribución adicional, al implementar la estrategia de natural rewriting para los módulos de sistema Maude. Esto es especialmente relevante porque es la primera estrategia bajo demanda implementada para este tipo de módulos, complementando los comandos Maude por defecto para reescritura y búsqueda en anchura.

A la vista de estos resultados podemos concluir que hemos hecho avances significativos en cada uno de nuestros objetivos originales. Sin embargo, como estos objetivos eran bastante generales, todavía queda mucho trabajo que hacer en esas líneas. En la siguiente sección esbozaremos algunas posibles extensiones de nuestro trabajo.

5.2 Trabajo futuro

Las siguientes son algunas posibles líneas abiertas para el trabajo futuro.

- *Estrategias bajo demanda para let-reescritura y let-estrechamiento*: Dichas nociones de reducción todavía no pueden ser consideradas un mecanismo operacional efectivo a un nivel práctico, debido a la ausencia de una estrategia de selección del redex que podría determinar para cada paso de reducción qué regla aplicar y sobre qué subexpresión. Existen varias estrategias de evaluación bajo demanda para reescritura y estrechamiento, siendo el estrechamiento necesario [AEH94] y el natural narrowing [EMT05] dos de las más conocidas en el campo de la FLP. En la práctica el uso de alguna estrategia es indispensable porque en otro caso el espacio de búsqueda de los cómputos crece hasta volverse inmanejable.

  Nuestras nociones de reducción fueron concebidas, por un lado para ser independientes de una estrategia bajo demanda en particular, y por otro para facilitar la adopción de cualquier estrategia concreta. Por tanto consideramos que la ausencia de una estrategia implícita es una ventaja en lugar de un defecto. En algún trabajo futuro podríamos investigar el uso de estrategias para desarrollar relaciones derivadas

de la *let*-reescritura y el *let*-estrechamiento, que serían subrelaciones de éstas, ya que las estrategias se emplean para reducir el espacio de búsqueda al evitar pasos innecesarios. Consideramos que la *let*-reescritura podría ser precisamente la herramienta adecuada para demostrar la optimalidad y adecuación de estas estrategias cuando se usan en un entorno call-time choice, lo que sigue siendo un problema abierto, ya que estas estrategias fueron formuladas originalmente para la reescritura de términos, es decir, para run-time choice.

- *Extensiones de la let-reescritura*: Ya hemos extendido el marco de la *let*-reescritura al entorno de orden superior establecido por *HOCRWL* [GHR97], por lo que creemos que el siguiente paso natural sería utilizarlo para resolver algunos problemas bien conocidos sobre sistemas de tipos que han surgido en la FLP. Ya ha habido algunos avances en esa linea, en particular en [LMR10] usamos la *HOlet*-reescritura y una extensión del sistema de tipos de Damas & Milner [DM82] para solucionar una violación de la propiedad de subject reduction relacionada con los patrones de orden superior.

  Otro problema complicado, ya mencionado en [LRS08a] y conocido desde hace mucho tiempo [GHR01], es el de las violaciones de la propiedad de subject reduction causadas por el estrechamiento con variables de orden superior, así como el consecuente crecimiento descontrolado del espacio de búsqueda. En [GHR01] un cálculo de resolución de objetivos extendido con información de tipos se propuso para resolver este problema, quizás algunas de esas ideas podrían ser adaptadas al marco del *HOlet*-estrechamiento.

  Finalmente, podríamos extender la *let*-reescritura y estrechamiento para manejar restricciones, que son otra característica común de los sistemas FLP.

- *Avances en nuestra lógica de reescritura para CS's*: La lógica desarrollada en [LRS09b] se diseñó con los CS's en mente, y por tanto utiliza la noción de constructora para definir su dominio denotacional. La disciplina de constructoras es una disciplina de programación muy aceptada, adoptada por la mayoría de los lenguajes funcionales, y en realidad los programas ecuacionales que violan esta disciplina de constructoras son raros en la práctica [O'D85]. Sin embargo, como consecuencia de usar un dominio basado en la noción de constructor, nuestra lógica no puede usarse con otras clases de TRS's que no sigan la disciplina de constructoras. Por tanto sería interesante tratar de superar estas limitaciones, reemplazando el papel de los valores basados en constructoras por alguna alternativa apropiada. Creemos que un enfoque prometedor consistiría en tomar las ideas de [Tha85, DS93, Sal95], donde se demuestra que cualquier TRS ortogonal puede ser transformado en un CS equivalente, y adaptarlas para definir un nuevo dominio denotacional que podríamos utilizar para definir una extensión de nuestra lógica para una clase más general de TRS's.

- *Comparación de descripciones semánticas del call-time choice*: Hasta ahora nuestra comparativa sólo cubre *CRWL*, la reescritura de términos y la semántica operacional de FLC de [AHH+05]. Además los resultados de equivalencia entre *CRWL* y FLC están dados para una clase restringida de programas. Para empezar nos gustaría extender estos resultados para cubrir la clase completa de programas *CRWL*. Quizás nuestra relación de *let*-reescritura podría ser una herramienta apropiada para esta tarea, ya que es una noción operacional y por tanto mucho más cercana a *FLC* que

CRWL. Por otra parte encontraríamos interesante demostrar la esperada equivalencia entre *CRWL* y la formalización de la reescritura de grafos de [EJ97, EJ98], que es también una importante familia de descripciones semánticas que ha sido utilizada en muchos trabajos sobre FLP, como por ejemplo en [ABC07, ABC06].

- *Comparación de semánticas para el indeterminismo con programas indeterministas*: En la sección 3.3.2 demostramos la equivalencia de *CRWL* y la reescritura de términos para la clase de programas deterministas, y conjeturamos fuertemente que la confluencia de un programa *CRWL* bajo reescritura implica su determinismo. Antes de nada nótese que el determinismo fue definido en términos de *CRWL*: un programa es determinista cuando para cualquier expresión su denotación *bajo CRWL* es un conjunto dirigido. Así que podemos concebir otras dos nociones de determinismo de los programas usando las denotaciones definidas o bien por la semántica para reescritura de la sección 3.3.1, o por la lógica $\pi CRWL$ de la sección 3.4.1. Demostrar la conjetura mencionada, estudiar las relaciones entre dichas nociones de determinismo, y finalmente estudiar la posible equivalencia de *CRWL*, reescritura y $\pi CRWL$ bajo diferentes suposiciones de determinismo son algunas posibles lineas de trabajo futuro abiertas en este tema.

- *El problema de la full abstraction*: Como ya se discutió en la sección 3.2.5, para lenguajes de orden superior bajo semántica de call-time choice sólo obtuvimos resultados positivos para programas sin variables extra, por lo que sería interesante tratar de superar estas limitaciones. Pensamos que esto sería posible extendiendo la sintaxis con lambda abstracciones, que son un antiguo tema pendiente para el marco *CRWL*. Aunque una variación de *CRWL* para soportar lambda abstracciones se propone en [Vad07, Vad09], esta sigue un enfoque diferente al de la lógica *HOCRWL* presentada en la sección 3.2.4 y originalmente desarrollada en [GHR97, GHR01], porque las nociones de programa y valor empleadas allí son fundamentalmente diferentes a las empleadas en *HOCRWL*. La lógico propuesta en [Vad07, Vad09] describe la semántica de un lenguaje esencialmente diferente a los sistemas FLP populares como *Toy* o *Curry*, mientras que *HOCRWL* proporciona una descripción para un subconjunto de estos lenguajes, en particular no incluyendo soporte para lambda abstracciones. La adición de lambda abstracciones a estos lenguajes es también una tarea pendiente para toda la comunidad FLP, porque aunque algunos sistemas FLP ofrecen soporte para lambda abstracciones [Lux07, Han09], estas tienen comportamientos diferentes en los distintos sistemas. Por tanto una exploración del espacio de diseño para la introducción de lambda abstracciones en sistemas como *Toy* o *Curry*, dando caracterizaciones formales de las diferentes alternativas y estudiando sus propiedades, sería interesante en cualquier caso.

  Otra extensión ortogonal de nuestros resultados acerca de la full abstraction podría ir en la linea de dar un papel más activo a las variables, que hasta ahora han sido tratadas en nuestros trabajos prácticamente como constantes. Esto podría ser interesante teniendo en cuenta que las variables pueden ser instanciadas por estrechamiento, o simplemente por la instanciación causada por el paso de parámetros de la reescritura. Creemos que esta extensión podría ser una herramienta importante para la transformación de programas, ofreciendo una condición necesaria importante para el reemplazamiento de lados derechos de reglas de programa.

- *Demostración de teoremas mecanizada*: Como ya comentamos antes, solamente hemos

dado los primeros pasos en esta dirección. Una posible extensión de nuestro trabajo podría ser tratar de combinar nuestros resultados sobre la meta-teoría con otros trabajos previos sobre *CRWL* e Isabelle [CLLF04], que se centraban en demostrar propiedades de programas concretos.

Otro posible tema de trabajo futuro en esta linea podría ser la formalización de la relación de *let*-reescritura en Isabelle para entonces demostrar su adecuación respecto a *CRWL* en este sistema. Este sería un paso en la dirección del reto 3 de [ABF+05], "Testing and Animating wrt. the Semantics", porque acabaríamos obteniendo un intérprete de *CRWL* durante el proceso. También deberíamos entonces formalizar alguna estrategia de evaluación bajo demanda para nuestra formalización en Isabelle de la *let*-reescritura, para obtener así una demostración en Isabelle de su optimalidad.

También podríamos hacer algo similar con nuestra semántica para sistemas de constructoras de la sección 3.3.1. Su formalización en Isabelle debería ser similar a la formalización de *CRWL*, y creemos que la implementación en Isabelle de su relación operacional asociada, es decir, de la reescritura de términos, sería también muy fácil, ya que es una noción bastante más simple que la *let*-reescritura, y cuya formalización en Isabelle ya ha sido abordada en la biblioteca IsaFoR (Isabelle Formalization of Rewriting) que es parte del sistema CeTA (Certified Termination Analysis) [TS09].

- *Combinaciones semánticas*: En [LRS09a] ya apuntamos una posible aplicación de nuestra combinación de call-time choice y run-time choice en un entorno call-time. La cuestión es que la mayoría de los sistemas FLP implementan la compartición de subexpresiones por medio de una construcción interna llamada suspensión, que juega un papel paralelo al de las ligaduras *let* de la *rt-let*-reescritura. Por tanto pensamos que este podría ser el nivel de abstracción más apropiado para razonar sobre las suspensiones, de cara a minimizar su creación y por tanto mejorar el rendimiento de los sistemas.

  Por otra parte la combinación de call-time choice y la semántica plural expresada por $\pi CRWL$ merece ser investigada, ya que ambas semánticas son composicionales con respecto a los c-términos, de esta manera permitiendo un estilo de programación basado en valores. Trataremos de nuevo con esta cuestión en el siguiente apartado.

- *Semántica plural*: Hay mucho trabajo que hacer acerca de $\pi CRWL$. Para empezar sus capacidades expresivas deberían ser estudiadas a fondo de cara a intentar obtener más patrones de programación interesantes que pudieran explotar las capacidades de esta nueva semántica. Ya hemos hecho algunos avances en esta dirección en [RR10]. En ese trabajo se hace aparente que la combinación de argumentos plurales y singulares en el mismo programa hace más fácil escribir los programas que en una semántica plural monolítica, y que al mismo tiempo el uso de argumentos plurales en algunos fragmentos del programa surge de forma natural y ayuda a mejorar el carácter declarativo de los programas.

  Entonces una noción operacional al nivel de la *let*-reescritura debería ser diseñada para $\pi CRWL$, ya que la implementación transformacional actual realiza mucha duplicación de cómputos. Cierta compartición de conjuntos de cómputos en la línea de [BH07] debería realizarse en dicha noción de reducción. Otro objetivo interesante sería diseñar un nuevo procedimiento de estrechamiento que fuera completo para $\pi CRWL$, ya que las relaciones de estrechamiento actuales son completas sólo para

sustituciones normalizantes, una condición que no cumplen las sustituciones usadas en $\pi CRWL$ para el paso de parámetros. Otras posibles extensiones del marco podrían incluir el soporte de variables extra en $\pi CRWL$ y la adición de características de orden superior o incluso capacidades de encaje de patrones módulo axiomas—un rasgo prominente de la lógica de reescritura de Meseguer [MM02], que se implementa en el sistema Maude [CDE+07].

Finalmente la noción de $SCTerm$ de [LRS09b] parece ser una noción de valor más natural para $\pi CRWL$. Aunque fuimos capaces de formular esta lógica usando c-términos únicamente, algunas dificultades para expresar propiedades como el cierre bajo sustituciones podrían ser tratadas de forma natural usando $SCTerm$'s. Esto nos sugiere un posible marco semántico unificado en el que las semánticas de call-time choice, run-time choice y $\pi CRWL$, y cualquier combinación de estas, pudieran ser expresadas por medio de una única lógica.

- *Alternativas semánticas para el lambda cálculo indeterminista*: En [KSS98] se presenta una extensión indeterminista del lambda cálculo. Este marco fue extendido en [SSH00] con constructoras y primitivas *letrec* y *case*. Esto nos proporciona los ingredientes necesarios para trasladar la jerarquía semántica establecida para sistemas de reescritura indeterministas al mundo del lambda cálculo indeterminista. Encontramos esta línea bastante interesante también porque las técnicas usadas en estos trabajos son fundamentalmente diferentes a las nuestras, al estar basadas en las nociones de equivalencia observacional y bisimulación. Además en estos trabajos hay una preocupación por la dimensión demoníaca/angélica/errática del indeterminismo, algo que no ha sido considerado en nuestros trabajos, que se centran en indeterminismo angélico únicamente. Por ello consideramos que tratar de transferir ideas entre estas dos líneas de investigación podría ser provechoso para ambas.

- *Aspectos algorítmicos del indeterminismo*: Como mencionamos en la introducción, varios trabajos en el campo de la programación funcional han sido desarrollados para tratar de simular el indeterminismo por medio de diferentes construcciones [Wad85, Hin00, KSFS05, FBK05, NAR07, FKS09]. Nosotros también hicimos una pequeña contribución al tema en [LRS07c]. Sería interesante para nosotros profundizar en esta linea, intentando obtener un procedimiento más sistemático para definir estructuras de datos para representar el indeterminismo, pero también intentando representar no sólo la semántica de call-time choice habitual presentada en trabajos previos, sino también las diferentes semánticas presentadas en esta tesis, así como sus posibles combinaciones.

Part II

Publications Associated to the Thesis

Chapter 6

List of Publications

6.1 First Level Publications

[LRS07b] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Simple Rewrite Notion for Call-time Choice Semantics. In *Proceedings of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM, 2007.

[LRS08a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Rewriting and Call-Time Choice: The HO Case. In *Proceedings of the 9th Fuji International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2008.

[Rod08] Juan Rodríguez-Hortalá. A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems. In *Proceedings of the 28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 328–339, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[LRS09a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Flexible Framework for Programming with Non-deterministic Functions. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, pages 91–100. ACM, 2009.

[LRS09b] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Fully Abstract Semantics for Constructor Systems. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2009.

6.2 Other Publications

[LRS09d] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Narrowing for First Order Functional Logic Programs with Call-Time Choice Semantics. In *Proceedings of the 17th International Conference on Applications of Declar-*

ative Programming and Knowledge Management (INAP'07) and 21st Workshop on (Constraint) Logic Programming (WLP'07), Revised Selected Papers, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 206–222. Springer, 2009.

[LRS07a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Equivalence of Two Formal Semantics for Functional Logic Programs. *Electronic Notes in Theoretical Computer Science*, 188:117–142, 2007. Proceedings of the Sixth Spanish Conference on Programming and Languages (PROLE'06).

[RR09b] Adrián Riesco and Juan Rodríguez-Hortalá. A natural implementation of Plural Semantics in Maude (demonstration). *Electronic Notes in Theoretical Computer Science*, 2009. To appear in Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA'09).

[LRS09c] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Lightweight Combination of Semantics for Non-deterministic Functions. *Computing Research Repository (CoRR)*, abs/0903.2205, 2009. Proceedings of the 18th Workshop on Logic-based methods in Programming Environments (WLPE'08).

[LR10] Francisco Javier López-Fraguas and Juan Rodríguez-Hortalá. The Full Abstraction Problem for Higher Order Functional-Logic Programs. *Computing Research Repository CoRR*, abs/1002.1833, 2010. Proceedings of the 19th Workshop on Logic-based methods in Programming Environments (WLPE'09).

[LMR09a] Francisco Javier López-Fraguas, Stephan Merz, and Juan Rodríguez-Hortalá. A Formalization of the Semantics of Functional-Logic Programming in Isabelle. *Computing Research Repository CoRR*, abs/0908.0494, 2009. Emerging trends section (category B) of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09).

Chapter 7

Publications (full text)

7.1 First level publications

7.1.1 A Simple Rewrite Notion for Call-time Choice Semantics

Available online at http://portal.acm.org/citation.cfm?id=1273947.

7.1.2 Rewriting and Call-time Choice: The HO Case

Available online at http://www.springerlink.com/content/h882302u4881838q.

7.1.3 A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems

Foundations of Software Technology and Theoretical Computer Science (Bangalore) 2008.
Editors: R. Hariharan, M. Mukund, V. Vinay; pp 328-339

A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems*

Juan Rodríguez-Hortalá
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
juanrh@fdi.ucm.es

ABSTRACT. Formalisms involving some degree of nondeterminism are frequent in computer science. In particular, various programming or specification languages are based on term rewriting systems where confluence is not required. In this paper we examine three concrete possible semantics for non-determinism that can be assigned to those programs. Two of them –call-time choice and run-time choice– are quite well-known, while the third one –plural semantics– is investigated for the first time in the context of term rewriting based programming languages. We investigate some basic intrinsic properties of the semantics and establish some relationships between them: we show that the three semantics form a hierarchy in the sense of set inclusion, and we prove that call-time choice and plural semantics enjoy a remarkable compositionality property that fails for run-time choice; finally, we show how to express plural semantics within run-time choice by means of a program transformation, for which we prove its adequacy.

1 Introduction

Term rewriting systems (*TRS's*) [4] have a long tradition as a suitable basic formalism to address a wide range of tasks in computer science, in particular, many specification languages [5, 7], theorem provers [21, 6] and programming languages are based on TRS's. For instance, the syntax and theory of TRS's was the basis of the first formulations of *functional logic programming* (*FLP*) through the idea of narrowing [9]. On the other hand, non-determinism is an expressive feature that has been used for a long time in system specification (e.g., non-deterministic Turing machines or automata) or for programming (the constructions of McCarthy and Dijkstra are classical examples). One of the appeals of term rewriting is its elegant way to express non-determinism through the use of a non-confluent TRS, obtaining a clean and high level representation of complex systems. In the field of FLP, non-confluent TRS's are used as programs to support non-strict non-deterministic functions, which are one of the most distinctive features of the paradigm [8, 3]. Those TRS's follow the constructor discipline also, corresponding to a value-based semantic view, in which the purpose of computations is to produce values made of constructors.

Therefore non-confluent constructor-based TRS's can be used as a common syntactic framework for FLP and rewriting. The set of rewrite rules constitutes a program and so we also call them *program rules*. Nevertheless the behaviour of current implementations of FLP

*This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), Promesas-CAM (S-0505/TIC/0407) and FAST-STAMP (TIN2008-06622-C03-01/TIN).

FSTTCS 2008
IARCS Annual Conference on
Foundations of Software Technology and Theoretical Computer Science
http://drops.dagstuhl.de/opus/volltexte/2008/1764

and rewriting differ substantially, because the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions, that were explored in [20]. There the different language variants that result after adding non-determinism to a basic functional language were expounded, structuring the comparison as a choice among different options over several dimensions: strict/non-strict functions, angelic/demonic/erratic non-deterministic choices and *singular/plural semantics* for parameter passing. In the present paper we assume non-strict angelic non-determinism, and we are concerned about the last dimension only. The key difference is that under a singular semantics, in the substitutions used to instantiate the program rules for function application, the variables of the program rules should range over single objects of the set of values considered; in a plural semantics those range over sets of objects. This has been traditionally identified with the distinction between *call-time choice* and *run-time choice* [11] parameter passing mechanisms. Under call-time choice a value for each argument is computed before performing parameter passing, this corresponds to call-by-value in a strict setting and to call-by-need in a non-strict setting, in which a partial value instead of a total value is computed. On the other hand, run-time-choice corresponds to call-by-name, each argument is copied without any evaluation and so the different copies of any argument may evolve in different ways afterwards. Thus, traditionally it has been considered that call-time choice parameter passing inducts a singular semantics while run-time choice inducts a plural semantics.

EXAMPLE 1. *Consider the TRS* $\mathcal{P} = \{f(c(X)) \to d(X,X), X\ ?\ Y \to X, X\ ?\ Y \to Y\}$. *With call-time choice/singular semantics to compute a value for the term* $f(c(0?1))$ *we must first compute a (partial) value for* $c(0?1)$*, and then we may continue the computation with* $f(c(0))$ *or* $f(c(1))$ *which yield* $d(0,0)$ *or* $d(1,1)$. *Note that* $d(0,1)$ *and* $d(1,0)$ *are not correct values for* $f(c(0?1))$ *in that setting.*
On the other hand with run-time choice/plural semantics to evaluate the term $f(c(0?1))$*:*

- *Under the run-time choice point of view, the step* $f(c(0?1)) \to d(0?1,0?1)$ *is sound, hence not only* $d(0,0)$ *and* $d(1,1)$ *but also* $d(0,1)$ *and* $d(1,0)$ *are valid values for* $f(c(0?1))$.
- *Under the plural semantics point of view, we consider the set* $\{c(0),c(1)\}$ *which is a subset of the set of values for* $c(0?1)$ *in which every element matches the argument pattern* $c(X)$. *Therefore the set* $\{0,1\}$ *can be used for parameter passing obtaining a kind of "set expression"* $d(\{0,1\},\{0,1\})$*, which evaluation yields the values* $d(0,0)$*,* $d(1,1)$*,* $d(0,1)$ *and* $d(1,0)$.

In general, call-time choice/singular semantics produces less results than run-time choice/plural semantics.

A standard formulation for call-time choice[†] in FLP is the *CRWL*[‡] logic [8], which is implemented by current FLP languages like Toy [15] or Curry [10]; traditional term rewriting may be considered the standard semantics for run-time choice[§], and is the basis for the semantics of languages like Maude [5], but has been rarely [1] thought as a valuable global alternative to call-time choice for the value-based view of FLP. However, there might be

[†] In fact angelic non-strict call-time choice.
[‡] **C**onstructor-based **Re**ducibility **W**riting **L**ogic.
[§] In fact angelic non-strict run-time choice.

parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal [13]. Nevertheless the use of an operational notion like term rewriting as the semantic basis of a FLP language can lead us to confusing situations, not very compatible with the value-based semantic view that we wanted for the constructor-based TRS's used in FLP.

EXAMPLE 2. *Starting with the TRS of Example 1 we want to evaluate the expression* $f(c(0) ? c(1))$ *with run-time choice/plural semantics:*

- *Under the run-time choice point of view, that is, using term rewriting, the evaluation of the subexpression* $c(0)?c(1)$ *is needed in order to get an expression that matches the left hand side* $f(c(X))$*. Hence the derivations* $f(c(0)?c(1)) \to f(c(0)) \to d(0,0)$ *and* $f(c(0)?c(1)) \to f(c(1)) \to d(1,1)$ *are sound and compute the values* $d(0,0)$ *and* $d(1,1)$*, but neither* $d(0,1)$ *nor* $d(1,0)$ *are correct values for* $f(c(0)?c(1))$*.*
- *Under the plural semantics point of view, we consider the set* $\{c(0), c(1)\}$ *which is a subset of the set of values for* $c(0)?c(1)$ *in which every element matches the argument pattern* $c(X)$*. Therefore the set* $\{0,1\}$ *can be used for parameter passing obtaining a kind of "set expression"* $d(\{0,1\},\{0,1\})$ *that yields the values* $d(0,0)$*,* $d(1,1)$*,* $d(0,1)$ *and* $d(1,0)$*.*

Which of these is the more suitable perspective for FLP?

This problem did not appear in [20] because no pattern matching was present, nor in [11] because only call-time choice was adopted (and this conflict does not appear between the call-time choice and the singular semantics views). Choosing the run-time choice perspective of term rewriting has some unpleasant consequences. First of all the expression $f(c(0?1))$ has more values than the expression $f(c(0)?c(1))$, even when the only difference between them is the subexpressions $c(0?1)$ and $c(0)?c(1)$, which have the same values both in call-time choice, run-time choice and plural semantics. This is pretty incompatible with the value-based semantic view we are looking for in FLP. And this has to do with the loss of some desirable properties, present in *CRWL*, when switching to run-time choice. We will see how plural semantics recovers those properties, which are very useful for reasoning about computations. Furthermore it allows natural encodings of some programs that need to do some collecting work, as we will see later (Example 8). Hence we claim that the plural semantics perspective is more suitable for a value-based programming language.

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about *CRWL* and term rewriting systems. In Section 3 we introduce π*CRWL*, a variation of *CRWL* to express plural semantics, and present some of its properties. In Section 4 we discuss about the different properties of these semantics and prove the inclusion chain *CRWL* $\subseteq$ rewriting $\subseteq$ π*CRWL*, that constitutes a hierarchy of semantics for non-determinism. Section 5 recalls that no straight simulation of *CRWL* in term rewriting can be done by a program transformation, and vice versa, and shows a novel transformation to simulate π*CRWL* using term rewriting. Finally Section 6 summarizes some conclusions and future work. Fully detailed proofs, including some auxiliary results, can be found in [19].

2 Preliminaries

2.1 Constructor based term rewriting systems

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\bar{o}$ stands for tuples of any kind of syntactic objects. Given a set $\mathcal{A}$ we denote by $\mathcal{A}^*$ the set of finite sequences of elements of that set. For any sequence $a_1 \ldots a_n \in \mathcal{A}^*$ and function $f : \mathcal{A} \to \{true, false\}$, by $a_1 \ldots a_n \mid f$ we denote the sequence constructed taking in order every element from $a_1 \ldots a_n$ for which f holds.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing **values**. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ϵ for the identity (or empty) substitution. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some θ_i; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. Given $W \subseteq \mathcal{V}$ we write $\theta|_W$ for the restriction of θ to W, and $\theta|_{\setminus D}$ is a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$.

A *constructor-based term rewriting system* $\mathcal{P}$ (*CS*) is a set of c-rewrite rules of the form $f(\bar{t}) \to r$ where $f \in FS^n$, $e \in Exp$ and $\bar{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\bar{t}$. In the present work we restrict ourselves to CS's not containing *extra variables*, i.e., CS's for which $var(r) \subseteq var(f(\bar{t}))$ holds for any rewrite rule; the extension of this work to rules with extra variables is a subject of future work. We assume that every CS $\mathcal{P}$ contains the rules $\{X \,?\, Y \to X, X \,?\, Y \to Y, if\ true\ then\ X \to X\}$, defining the behaviour of $\_?\_ \in FS^2$, $if\_then\_ \in FS^2$, both used in mixfix mode, and that those are the only rules for that function symbols. For the sake of conciseness we will often omit these rules when presenting a CS.

Given a TRS $\mathcal{P}$, its associated *rewrite relation* $\to_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\sigma] \to_{\mathcal{P}} \mathcal{C}[r\sigma]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\sigma \in Subst$. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$ or denote it by $\mathcal{P} \vdash e \to e'$ and $\mathcal{P} \vdash e \to^* e'$.

$$\textbf{(RR)}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad \textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\quad c \in CS^n$$

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(OR)}\ \frac{e_1 \twoheadrightarrow p_1\theta \ldots e_n \twoheadrightarrow p_n\theta \;\; r\theta \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t}\quad \begin{array}{l} f(p_1,\ldots,p_n) \to r \in \mathcal{P} \\ \theta \in CSubst_\bot \end{array}$$

Figure 1: Rules of *CRWL*

2.2 The CRWL framework

In the *CRWL* framework [8], programs are CS's, also called *CRWL-programs* (or simply 'programs') from now on. To deal with non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot$, $CTerm_\bot$, $Subst_\bot$, $CSubst_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, meaning informally that t is (or approximates to) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice. The *CRWL*-proof calculus is presented in Figure 1. Rule **B** (bottom) allows any expression to be undefined or not evaluated (non-strict semantics). Rule **OR** (outer reduction) expresses that to evaluate a function call we must choose a compatible program rule, perform parameter passing (by means of a $CSubst_\bot$ θ) and then reduce the right-hand side. The use of partial c-substitutions in OR is essential to express call-time choice, as only single partial values are used for parameter passing.

We write $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in the *CRWL*-calculus using the program $\mathcal{P}$. Given a program $\mathcal{P}$, the *CRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{sg}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$. In the following, we will usually omit the reference to $\mathcal{P}$.

3 π*CRWL*: a plural semantics for FLP

The new calculus π*CRWL* is defined by modifying the rules of *CRWL* to consider sets of partial values for parameter passing instead of single partial values: hence, only the rule OR should be modified. To avoid the need to extend the syntax with new constructions to represent those "set expressions" that we talked about in the introduction, we will exploit the fact that $[\![e_1 \;?\; e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$. Therefore the substitutions used for parameter passing will map variables to "disjunctions of values". We define the set $CSubst^?_\bot = \{\theta \in Subst_\bot \mid \forall X \in dom(\theta), \theta(X) = t_1 \;?\; \ldots \;?\; t_n$ such that $t_1, \ldots, t_n \in CTerm_\bot, n > 0\}$, for which $CSubst_\bot \subseteq CSubst^?_\bot \subseteq Subst_\bot$ obviously holds. The operator $? : CSubst^*_\bot \to CSubst^?_\bot$ constructs the $CSubst^?_\bot$ corresponding to a non empty sequence of $CSubst_\bot$, and is defined as $?(\theta_1 \ldots \theta_n)(X) = X$ if $X \notin \bigcup_{i \in \{1,\ldots,n\}} dom(\theta_i)$; $?(\theta_1 \ldots \theta_n)(X) = \rho_1(X) \;?\; \ldots \;?\; \rho_m(X)$, where $\rho_1 \ldots \rho_m = \theta_1 \ldots \theta_n \mid \lambda\theta.(X \in dom(\theta))$, otherwise. Then $dom(?(\theta_1 \ldots \theta_n)) = \bigcup_i dom(\theta_i)$. This

$$\textbf{(RR)}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad \textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\quad c \in CS^n$$

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(POR)}\ \frac{\begin{array}{ccc} e_1 \twoheadrightarrow p_1\theta_{11} & & e_n \twoheadrightarrow p_n\theta_{n1} \\ \ldots & \ldots & \ldots \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & e_n \twoheadrightarrow p_n\theta_{nm_n} \quad r\theta \twoheadrightarrow t \end{array}}{f(e_1,\ldots,e_n) \twoheadrightarrow t}$$

$$(f(\overline{p}) \to r) \in \mathcal{P}, \theta = ?\{\theta_{11},\ldots,\theta_{1m_1}\} \uplus \ldots \uplus ?\{\theta_{n1},\ldots,\theta_{nm_n}\}$$
$$\forall i,j\ \theta_{ij} \in CSubst_\bot \wedge dom(\theta_{ij}) = var(p_i), \forall i\ m_i > 0$$

Figure 2: Rules of $\pi CRWL$

operator is overloaded to handle non empty sets $\Theta \subseteq CSubst_\bot$ as $?\Theta = ?(\theta_1 \ldots \theta_n)$ where the sequence $\theta_1 \ldots \theta_n$ corresponds to an arbitrary reordering of the elements of Θ.

The $\pi CRWL$-proof calculus is presented in Figure 2. The only difference with the calculus in Figure 1 is that the rule OR has been replaced by **POR** (plural outer reduction), in which we may compute more that one partial value for each argument, and then use a substitution from $CSubst^?_\bot$ instead of $CSubst_\bot$ for parameter passing, achieving a plural semantics¶. This calculus derives reduction statements of the form $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ that express that t is (or approximates to) a possible value for e in this semantics, under the program $\mathcal{P}$. The $\pi CRWL$-*denotation* of an expression $e \in Exp_\bot$ under a program $\mathcal{P}$ in $\pi CRWL$ is defined as $[\![e]\!]^{pl}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t\}$.

EXAMPLE 3. *Consider the program of Example 1, that is* $\mathcal{P} = \{f(c(X)) \to d(X,X), X\,?\,Y \to X, X\,?\,Y \to Y\}$. *The following is a* $\pi CRWL$*-proof for the statement* $f(c(0)?c(1)) \twoheadrightarrow d(0,1)$ *(some steps have been omitted for the sake of conciseness):*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \twoheadrightarrow 0}\,DC}{c(0) \twoheadrightarrow c(0)}\,DC \quad \dfrac{}{c(1) \twoheadrightarrow \bot}\,B}{c(0)?c(1) \twoheadrightarrow c(0)}\,POR \quad c(0)?c(1) \twoheadrightarrow c(1) \quad \dfrac{0?1 \twoheadrightarrow 0 \quad 0?1 \twoheadrightarrow 1}{d(0?1,0?1) \twoheadrightarrow d(0,1)}\,DC}{f(c(0)?c(1)) \twoheadrightarrow d(0,1)}\,POR$$

$\pi CRWL$ enjoys some nice properties, like the following monotonicity property, where for any proof we define its *size* as the number of applications of rules of the calculus.

LEMMA 4. *For any CRWL-program,* $e, e' \in Exp_\bot$, $t, t' \in CTerm_\bot$ *if* $e \sqsubseteq e'$ *and* $t' \sqsubseteq t$ *then* $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ *implies* $\mathcal{P} \vdash_{\pi CRWL} e' \twoheadrightarrow t'$ *with a proof of the same size or smaller.*

One of the most important properties is its compositionality, a property very close to the DET-additivity property for algebraic specifications of [11]:

THEOREM 5. *For any CRWL-program,* $\mathcal{C} \in Contx$ *and* $e \in Exp_\bot$, $[\![\mathcal{C}[e]]\!]^{pl} = \bigcup_{\{t_1,\ldots,t_n\} \subseteq [\![e]\!]^{pl}} [\![\mathcal{C}[t_1\ ?\ \ldots\ ?\ t_n]]\!]^{pl}$, *for any arrangement of the elements of* $\{t_1,\ldots,t_n\}$ *in* $t_1\ ?\ \ldots\ ?\ t_n$.

The proof for that theorem is based upon the commutativity, associativity of ?, and the idempotence of its partial application (see [19]). With Theorem 5 at hand is very easy to prove the following distributivity property for $\pi CRWL$, also known as the *bubbling* operational rule [2]:

¶In fact angelic non-strict plural non-determinism.

THEOREM 6. *[Correctness of bubbling] For any CRWL-program, $\mathcal{C} \in Contx$ and $e_1, e_2 \in Exp_\bot$, $[\![\mathcal{C}[e_1 \,?\, e_2]]\!]^{pl} = [\![\mathcal{C}[e_1] \,?\, \mathcal{C}[e_2]]\!]^{pl}$.*

π*CRWL* also has some monotonicity properties related to substitutions. We define the preorder $\sqsubseteq_\pi$ over $CSubst^?_\bot$ by $\theta \sqsubseteq_\pi \theta'$ iff $\forall X \in \mathcal{V}$, given $\theta(X) = t_1 \,?\, \ldots \,?\, t_n$ and $\theta(X) = t'_1 \,?\, \ldots \,?\, t'_m$ then $\forall t \in \{t_1, \ldots, t_n\} \exists t' \in \{t'_1, \ldots, t'_m\}$ such that $t \sqsubseteq t'$; and the preorder $\trianglelefteq$ over $Subst_\bot$ by $\sigma \trianglelefteq \sigma'$ iff $\forall X \in \mathcal{V}$, $[\![\sigma(X)]\!]^{pl} \subseteq [\![\sigma'(X)]\!]^{pl}$.

LEMMA 7. *For any CRWL-program, $e \in Exp_\bot$, $t \in CTerm_\bot$, $\sigma, \sigma' \in Subst_\bot$, $\theta, \theta' \in CSubst^?_\bot$:*

1. ***Strong monotonicity of $Subst_\bot$**: If $\forall X \in \mathcal{V}, s \in CTerm_\bot$ given $\mathcal{P} \vdash_{\pi CRWL} \sigma(X) \twoheadrightarrow s$ with size K we also have $\mathcal{P} \vdash_{\pi CRWL} \sigma'(X) \twoheadrightarrow s$ with size $K' \leq K$, then $\vdash_{\pi CRWL} e\sigma \twoheadrightarrow t$ with size L implies $\vdash_{\pi CRWL} e\sigma' \twoheadrightarrow t$ with size $L' \leq L$.*
2. ***Monotonicity of $CSubst_\bot$**: If $\theta, \theta' \in CSubst_\bot$ and $\theta \sqsubseteq \theta'$ then $\mathcal{P} \vdash_{\pi CRWL} e\theta \twoheadrightarrow t$ with size K implies $\mathcal{P} \vdash_{\pi CRWL} e\theta' \twoheadrightarrow t$ with size $K' \leq K$.*
3. ***Monotonicity of $Subst_\bot$**: If $\sigma \trianglelefteq \sigma'$ then $[\![e\sigma]\!]^{pl} \subseteq [\![e\sigma']\!]^{pl}$.*
4. ***Monotonicity of $CSubst^?_\bot$**: If $\theta \sqsubseteq_\pi \theta'$ then $[\![e\theta]\!]^{pl} \subseteq [\![e\theta']\!]^{pl}$.*

We end this section with an example of the use of π*CRWL* to model problems in which some collecting work has to be done.

EXAMPLE 8. *We want to represent the database of a bank in which we hold some data about its employees, this bank has several branches and we want to organize the information according to them. So we define a non-deterministic function branches to represent the set of branches: a set is identified then with a non-deterministic expression. In this line we define a non-deterministic function employees which conceptually returns the set of records containing the information regarding the employees that work in a branch. Now, to search for the names of two clerks we define the function twoclerks which is based upon find, which forces the desired pattern $e(N, S, clerk)$ over the set defined by $employees(branches)$.*

$\mathcal{P} = \{branches \to madrid, branches \to vigo, employees(madrid) \to e(pepe, men, clerk), employees(madrid) \to e(paco, men, boss), employees(vigo) \to e(maria, women, clerk), employees(vigo) \to e(jaime, women, boss), twoclerks \to find(employees(branches)), find(e(N, S, clerk)) \to (N, N)\}$

With term rewriting $twoclerks \to find(employees(branches)) \not\twoheadrightarrow^* (pepe, maria)$, because in that expression the evaluation of branches is needed and so one of the branches must be chosen. On the other hand with π*CRWL (some steps have been omitted for the sake of conciseness):*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk)}\ POR \quad \dfrac{\dots}{(pepe \,?\, maria, pepe \,?\, maria) \twoheadrightarrow (pepe, maria)}\ DC}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk) \,?\, e(maria, \bot, clerk)}\ POR}{find(employees(branches)) \twoheadrightarrow (pepe, maria)}\ POR}{twoclerks \twoheadrightarrow (pepe, maria)}\ POR$$

where

$$\dfrac{\dfrac{branches \twoheadrightarrow madrid}{}\ POR \quad \dfrac{\dots}{e(pepe, men, clerk) \twoheadrightarrow e(pepe, \bot, clerk)}\ DC}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk)}\ POR$$

4 Comparison: a hierarchy of semantics

When comparing these semantics is not surprising finding that *CRWL* and π*CRWL* have similar properties. For example the monotonicity Lemma 4 also holds for *CRWL*; this lemma

does not even make sense for term rewriting, as it only works with total terms. On the other hand term rewriting is closed under *Subst* ($e \to^* e'$ implies $e\sigma \to^* e'\sigma$, for any $\sigma \in Subst$); *CRWL* is not closed under *Subst* but under $CSubst_\perp$, as corresponds to call-time choice; π*CRWL* is closed under $CSubst_\perp$ too (see [19]), and we conjecture that for $\theta \in CSubst^?_\perp$ if $\vdash_{\pi CRWL} e \twoheadrightarrow t$ then $[\![t\theta]\!]^{pl} \subseteq [\![e\theta]\!]^{pl}$. For *CRWL* a compositionality result similar to Theorem 5 also holds, and bubbling is correct too [14]. This is not the case for term rewriting, as we saw when switching from $f(c(0?1))$ to $f(c(0)?c(1))$ in examples 1 and 2.

4.1 The hierarchy

As π*CRWL* is a modification of *CRWL*, the relation between them is very direct.

THEOREM 9. *For any CRWL-program $\mathcal{P}$, $e \in Exp_\perp, t \in CTerm_\perp$ given a CRWL-proof for $\mathcal{P} \vdash e \twoheadrightarrow t$ we can build a πCRWL-proof for $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ just replacing every **OR** step by the corresponding **POR** step. As a consequence $[\![e]\!]^{sg}_{\mathcal{P}} \subseteq [\![e]\!]^{pl}_{\mathcal{P}}$.*

Concerning the relation of *CRWL* and π*CRWL* with term rewriting, we will use the notion of *shell* $|e|$ *of an expression* e that represents the outer constructor (and thus computed) part of e, defined as $|\perp| = \perp$, $|X| = X$, $c(e_1,\ldots,e_n) = c(|e_1|,\ldots,|e_n|)$, $|f(e_1,\ldots,e_n)| = \perp$ (for $X \in \mathcal{V}, c \in CS, f \in FS$). We also define the denotation of $e \in Exp$ under term rewriting as $[\![e]\!]^{rw} = \{t \in CTerm_\perp \mid \exists e' \in Exp\ .\ e \to^* e' \wedge t \sqsubseteq |e'|\}$. In a previous joint work the author explored the relation between *CRWL* and term rewriting ([12], Theorem 9), recast in the following theorem:

THEOREM 10. *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rw}$. The converse inclusion does not hold in general.*

As we saw in Example 1, in general call-time choice semantics like *CRWL* produce less results than run-time choice semantics like the one induced by term rewriting. We will see that this kind of relation also holds for term rewriting and π*CRWL*.

THEOREM 11. *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{rw} \subseteq [\![e]\!]^{pl}$. The converse inclusion does not hold in general.*

The key for proving Theorem 11 is a lemma stating that $\forall e, e' \in Exp$ if $e \to e'$ then $[\![e']\!]^{pl} \subseteq [\![e]\!]^{pl}$, that is, that every rewriting step is sound wrt. π*CRWL*. The evident corollary for these theorems is the announced inclusion chain.

COROLLARY 12. *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rw} \subseteq [\![e]\!]^{pl}$. Hence $\forall t \in CTerm$, $\vdash_{CRWL} e \twoheadrightarrow t$ implies $e \to^* t$ which implies $\vdash_{\pi CRWL} e \twoheadrightarrow t$.*

5 Simulating plural semantics

In [12, 13] it was shown that neither *CRWL* can be simulated by term rewriting with a simple program transformation, nor vice versa. Nevertheless, plural semantics can be simulated by rewriting using the transformation presented in the current section, which could be used as the basis for a first implementation of π*CRWL* that we might use for experimentation. First we will present a naive version of this transformation, and show its adequacy; later we will propose some simple optimizations for it.

5.1 A simple transformation

DEFINITION 13. *Given a CRWL-program P, for every rule* $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ *such that* $f \notin \{\_?\_, if\_then\_\}$ *we define its transformation as:*

$$pST(f(p_1,\ldots,p_n) \to r) = f(Y_1,\ldots,Y_n) \to if\ match(Y_1,\ldots,Y_n)\ then\ r[\overline{X_{ij}/project_{ij}(Y_i)}]$$

- $\forall i \in \{1,\ldots,n\}, \{X_{i1},\ldots,X_{ik_i}\} = var(p_i) \cap var(r)$ *and* $Y_i \in \mathcal{V}$ *is fresh.*
- $match \in FS^n$ *fresh is defined by the rule* $match(p_1,\ldots,p_n) \to true$.
- Each $project_{ij} \in FS^1$ *is a fresh symbol defined by the single rule* $project_{ij}(p_i) \to X_{ij}$.

For $f \in \{\_?\_, if\_then\_\}$ *the transformation leaves its rules untouched.*

The function *match* is used to impose a "guard" that enforces the matching of each argument with its corresponding pattern. If we dropped this condition the translation of, for example, to rule $(null(nil) \to true)$, would be $(null(Y) \to true)$, which is clearly unsound as then $null(0 : nil) \to true$. Besides each pattern p_i has been replaced by a fresh variable Y_i, to which any expression can match, hence the arguments may be replicated freely by the rewriting process without demanding any evaluation and thus keeping its denotation untouched: this is the key to achieve completeness wrt. $\pi CRWL$. Later on, the functions $project_{ij}$ will just make the projection of each variable when needed.

EXAMPLE 14. *Applying this to Example 1 we get* $\{f(Y) \to if\ match(Y)\ then\ d(project(Y), project(Y)), match(c(X)) \to true, project(c(X)) \to X\}$ *, under which we can do:*

$\underline{f(c(0)?c(1))} \to if\ \underline{match(c(0)?c(1))}\ then\ d(project(c(0)?c(1)), project(c(0)?c(1)))$
$\to^* if\ true\ then\ d(\overline{project(c(0)?c(1))}, project(c(0)?c(1)))$
$\to d(\overline{project(\underline{c(0)?c(1)}), project(\underline{c(0)?c(1)})}) \to^* d(\underline{project(c(0))}, \underline{project(c(1))}) \to^* d(0,1)$

Concerning the adequacy of the transformation:

THEOREM 15. *For any CRWL-program* $\mathcal{P}$, $e \in Exp_\perp$ *built up on the signature of* $\mathcal{P}$, $[\![e]\!]^{pl}_{pST(\mathcal{P})} \subseteq [\![e]\!]^{pl}_{\mathcal{P}}$.

THEOREM 16. *For any CRWL-program* $\mathcal{P}$, $e \in Exp, t \in CTerm_\perp$ *built up on the signature of* $\mathcal{P}$, *if* $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ *then exists some* $e' \in Exp$ *built using symbols of the signature of* $pST(\mathcal{P})$ *such that* $pST(\mathcal{P}) \vdash e \to^* e'$ *and* $t \sqsubseteq |e'|$.

COROLLARY 17. *For any CRWL-program* $\mathcal{P}$, $e \in Exp$ *built using symbols of the signature of* $\mathcal{P}$, $[\![e]\!]^{pl}_{\mathcal{P}} = [\![e]\!]^{rw}_{pST(\mathcal{P})}$. *Hence* $\forall t \in CTerm\ \mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ *iff* $pST(\mathcal{P}) \vdash e \to^* t$.

5.2 An optimized transformation

Concerning the transformation, if a pattern is ground then no parameter passing will be done for it and so no transformation is needed: for $null(nil) \to true$ we get $\{null(Y) \to if\ match(Y)\ then\ true, match(nil) \to true\}$, which is equivalent. Besides, if the pattern is a variable then any expression matches it and the projection functions are trivial, so no transformation is needed neither, as happens with $pair(X) \to (X, X)$ for which $\{pair(Y) \to if\ match(Y)\ then\ (project(Y), project(Y)), match(X) \to true, project(X) \to X\}$ are returned.

DEFINITION 18. *Given a CRWL-program P, for every rule* $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ *we define its transformation as:*

$$pST(f(p_1,\ldots,p_n) \to r) = \begin{cases} f(p_1,\ldots,p_n) \to r & \text{if } \rho_1 \ldots \rho_m \text{ is empty} \\ f(\tau(p_1),\ldots,\tau(p_n)) \to \begin{array}{l} if\ match(Y_1,\ldots,Y_m) \\ then\ r[\overline{X_{ij}/project_{ij}(Y_i)}] \end{array} & \text{otherwise} \end{cases}$$

where $\rho_1 \ldots \rho_m = p_1 \ldots p_n \mid \lambda p.(p \notin \mathcal{V} \wedge var(p) \neq \emptyset)$.
- $\forall \rho_i$, $\{X_{i1},\ldots,X_{ik_i}\} = var(\rho_i) \cap var(r)$ *and* $Y_i \in \mathcal{V}$ *is fresh.*
- $\tau : CTerm \to CTerm$ *is defined by* $\tau(p) = p$ *if* $p \in \mathcal{V} \vee var(p) = \emptyset$ *and* $\tau(p) = Y_i$ *otherwise, for* $p \equiv \rho_i$.
- $match \in FS^m$ *fresh is defined by the rule* $match(\rho_1,\ldots,\rho_m) \to true$.
- Each $project_{ij} \in FS^1$ *is a fresh symbol defined by the single rule* $project_{ij}(\rho_i) \to X_{ij}$.

We will not give a formal proof for the adequacy of the optimization. Nevertheless note how this transformation leaves untouched the rules for _?_ and $if\_then\_$ without defining an special case for them. As the simple transformation worked well for that rules that suggests that we are doing the right thing. We end this section with an example application of the optimized transformation, over the program of Example 8:

EXAMPLE 19. *The only rule modified is the one for* $find$: $\{find(Y) \to if\ match(Y)\ then\ (project(Y), project(Y)), match(e(N,s,clerk)) \to true, project(e(N,s,clerk)) \to N\}$ *so:*

$\underline{twoclerks} \to \underline{find(employees(branches))}$
$\to if\ match(\overline{employees(branches)})\ then\ (project(employees(branches)), project(employees(branches)))$
$\to^* \underline{if\ match(\overline{e(pepe,men,clerk)})\ then\ (project(employees(branches)), project(employees(branches)))}$
$\to^* (project(\underline{employees(branches)}), project(\underline{employees(branches)}))$
$\to^* (\underline{project(e(pepe,men,clerk))}, \underline{project(e(\overline{maria,women,clerk}))} \to^* (pepe, maria)$

6 Conclusions

In this work we have pointed the different interpretations of run-time choice and plural semantics caused by pattern matching. To the best of our knowledge this distinction is stablished in the present paper for the first time, because in [20] no pattern matching was present and in [11] only call-time choice was adopted. We argue that the run-time choice semantics induced by term rewriting is not the best option for a value-based programming language like current implementations of FLP. For that context a plural semantics has been proposed for which the compositionality properties lost when turning from call-time choice to rewriting are recovered. Nevertheless, for other kind of rewriting based languages like Maude, which are not limited to constructor-based TRS's, term rewriting has been proven to be an effective formalism.

Our concrete contributions can be summarized as follows:
- We have presented the proof calculus $\pi CRWL$, a novel formulation of plural semantics for left-linear constructor-based TRS's, which are the kind of TRS's used in FLP. Some basic properties of the new semantics have been stated and proved, and by some examples we have shown how it allows natural encodings of some programs that need to do some collecting work (Sect. 3).

- We have compared the new calculus with *CRWL* and term rewriting, which are standard formulations for call-time choice and run-time choice respectively. The different properties of these calculi have been discussed and the inclusion chain $CRWL \subseteq$ rewriting $\subseteq \pi CRWL$ has been proved (Sect. 4).
- We have recalled some previous results about the impossibility of a straight simulation of *CRWL* in term rewriting or viceversa by a simple program transformation. Besides we have proposed a novel program transformation to simulate plural semantics with term rewriting, and proved its adequacy (Sect. 5).

From a practical point of view, it might be unrealistic to think that a monolithic semantic view is adequate for addressing all non-determinism present in a large program. In [13] we have started to investigate the combination of call-time choice and run-time choice in a unified framework. But as $\pi CRWL$ seems to be more suitable than run-time choice for a value-based language, we are planning to extend that work to plural semantics.

We contemplate other relevant subjects of future work:
- Extending the current results to programs with extra variables, that is, with rules $l \to r$ in which $var(r) \subseteq var(l)$ does not hold in general. We should also deal with conditional rules and equality constraints to cover the basic features of FLP languages.
- Studying the relation between the determinism of programs under *CRWL* [12] and $\pi CRWL$, which we conjecture is equivalent. We also conjecture that for deterministic programs $\forall e \in Exp, [\![e]\!]^{sg} = [\![e]\!]^{rw} = [\![e]\!]^{pl}$. Getting results about the relation of confluence and determinism of programs could be useful for analyzing the confluence of a TRS through its determinism. In the same line, the inclusion chain $CRWL \subseteq$ rewriting $\subseteq \pi CRWL$ could be used to study the termination of a TRS through its termination in *CRWL* and $\pi CRWL$.
- Developing a more operational rewrite notion for $\pi CRWL$ in the line of [12], which could be extended to narrowing like in [14]. A complexity study would be needed to ensure that the extra nondeterminism does not preclude the design of an efficient implementation. On the other hand the natural value for $\pi CRWL$ seems to be $\mathcal{P}(CTerm_\bot)$ instead of $CTerm_\bot$, a formulation in the line of [16] could be useful to forget about the tricky use of $\_?\_$.
- Finally, for the immediate future, it could be interesting implementing the transformation to simulate $\pi CRWL$ in some term rewriting based language like Maude [5]. Maybe the context-sensitive rewriting [18] features of Maude could be used to improve the laziness of the transformed program like in [17]. Besides, the matching-module capacities of Maude could be used to enhance the expressivity of plural semantics.

Acknowledgements: The author would like to thank Paco López Fraguas and Jaime Sánchez Hernández for their support and their useful suggestions. I would also like to thank the referees for their very valuable comments.

References

[1] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. ALP'97*, pages 16–30. Springer LNCS 1298, 1997.

[2] S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In Proc. Termgraph'06, pages 61–70. ENTCS, 176(1), 2007.

[3] S. Antoy and M. Hanus. Functional logic design patterns. In Proc. FLOPS'02, pages 67–87. Springer LNCS 2441, 2002.

[4] F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press, United Kingdom, 1998.
[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Springer LNCS 4350, 2007
[6] M. Clavel, M. Palomino, and A. Riesco. Introducing the itp tool: a tutorial. J. UCS 12(11), pages 1618–1650, 2006.
[7] R. Diaconescu and K. Futatsugi. An overview of CafeOBJ. ENTCS 15,1998.
[8] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. J. Log. Program. 40(1), pages 47–87, 1999.
[9] M. Hanus. The integration of functions into logic programming: From theory to practice. J. Log. Program. 19/20, pages 583–628, 1994.
[10] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
[11] H. Hussmann. Non-Determinism in Algebraic Specifications and Algebraic Programs. Birkhäuser Verlag, 1993.
[12] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In Proc. PPDP'07, pages 197–208. ACM Press, 2007.
[13] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions (Extended version). Tech. Rep. SIC-9-08, Universidad Complutense de Madrid, 2008. `http://gpd.sip.ucm.es/juanrh/pubs/tchrRTCT08.pdf`.
[14] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In Proc. FLOPS'08, pages 147–162. Springer LNCS 4989, 2008.
[15] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In Proc. RTA'99, pages 244–247. Springer LNCS 1631, 1999.
[16] F. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. TPLP, 4(1&2), pages 41–74, 2004.
[17] S. Lucas. Needed reductions with context-sensitive rewriting. In Proc. ALP/HOA'97, pages 129–143. Springer LNCS 1298, 1997.
[18] S. Lucas. Context-sensitive computations in functional and functional logic programs. J. Fun. Log. Program 1998(1), 1998.
[19] J. Rodríguez-Hortalá. A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems (Extended version). Tech. Rep. SIC-10-08, Universidad Complutense de Madrid, 2008. `http://gpd.sip.ucm.es/juanrh/pubs/tchrFSTTCS08.pdf`.
[20] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. The Computer Journal 35(5), pages 514–523, 1992.
[21] M. Wenzel. The isabelle/isar reference manual. `http://isabelle.in.tum.de/dist/Isabelle99-2/doc/isar-ref.pdf`.

7.1.4 A Flexible Framework for Programming with Non-deterministic Functions

Available online at http://portal.acm.org/citation.cfm?id=1480945.1480959.

7.1.5 A Fully Abstract Semantics for Constructor Systems

Available online at http://www.springerlink.com/content/j642411317674790/.

7.2 Other publications

7.2.1 Narrowing for First Order Functional Logic Programs with Call-Time Choice Semantics

Available online at http://www.springerlink.com/content/k28128067038q01m/.

7.2.2 Equivalence of Two Formal Semantics for Functional Logic Programs

Available online at http://linkinghub.elsevier.com/retrieve/pii/S1571066107004847.

7.2.3 A natural implementation of Plural Semantics in Maude (demonstration)

Available online at http://ldta.info/2009/ldta2009proceedings.pdf, to appear in Elsevier Electronic Notes in Theoretical Computer Science.

7.2.4 A Lightweight Combination of Semantics for Non-deterministic Functions

A Lightweight Combination of Semantics for Non-deterministic Functions *

Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. The use of non-deterministic functions is a distinctive feature of modern functional logic languages. The semantics commonly adopted is *call-time choice*, a notion that at the operational level is related to the *sharing* mechanism of lazy evaluation in functional languages. However, there are situations where *run-time choice*, closer to ordinary rewriting, is more appropriate. In this paper we propose an extension of existing call-time choice based languages, to provide support for run-time choice in localized parts of a program. The extension is remarkably simple at three relevant levels: syntax, formal operational calculi and implementation, which is based on the system *Toy*.

1 Introduction

Non-strict non-deterministic functions are a distinctive feature of modern functional logic languages (see [5] for a recent survey). It is known that the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions (see e.g. [12]). For term-rewriting based specifications, Hussmann [7] established a major distinction between *call-time choice* and *run-time choice*. Call-time choice is closely related to call-by-value and, in the case of strict semantics, it is easily implemented by innermost rewriting. In the case of non-strict semantics, things are more complicated, since the call-by-value view of call-time choice must include partial values. Operationally, this needs something similar to the sharing mechanism followed, for efficiency reasons, in (deterministic) functional languages under lazy evaluation. In contrast, run-time choice does not share, corresponds rather to call-by-name, and is realized by ordinary rewriting. For deterministic programs, run-time and call-time are able to produce the same set of values, but in general the set of values reachable by run-time choice is larger than that of call-time choice.

Non-deterministic functions with non-strict and call-time choice semantics were introduced in the functional logic setting with the *CRWL* framework [4],

* This work has been partially supported by the Spanish projects MERIT-FORMS-UCM (TIN2005-09207-C03-03), FAST-STAMP (TIN2008-06622-C03-01/TIN) and Promesas-CAM (S-0505/TIC/0407) .

in which programs are possibly non-confluent and non-terminating constructor-based term rewriting systems (*CTRS*). Since then, they are common part of daily programming in systems like *Curry* [6] or *Toy* [11]. Run-time choice has been rarely [1] considered as a valuable global alternative to call-time choice.

However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal. The following example illustrates the interest of combining both semantics.

Example 1. Modeling grammar rules for string generation can be directly done by CTRS like the following (non-confluent and non-terminating) one, in which we assume that texts (terminals) are represented as strings (lists of characters), that can be concatenated with $++$ (defined in a standard way):

$$letter \to \text{"a"} \;\; .... \;\; letter \to \text{"z"} \qquad word \to \text{""} \qquad word \to letter{+}{+}word$$

Disregarding syntax, this CTRS is a valid program in functional logic systems like *Curry* or *Toy*. The program acts as a non-deterministic generator of the texts in the language defined by the grammar. Each individual reduction leads to a string in the language.

The generation of palindromes (of even length, for simplicity) could be done by the rewrite rules:

$$palindrome \to palAux(word) \qquad palAux(X) \to X \mathbin{++} reverse(X)$$

where *reverse* is defined in any standard way. It is important to remark that the definition of *palindrome/palAux* works fine only if call-time choice is adopted for non-determinism, meaning operationally that in the (partial) reduction

$$palindrome \to palAux(word) \to word \mathbin{++} reverse(word)$$

the two occurrences of *word* created by the rule of *palAux* must be shared. If run-time choice (i.e., ordinary rewriting) were used, the two occurrences of *word* could follow independent ways, and therefore *palindrome* could be reduced, for instance, to *"oops"*, which is not a palindrome. Two useful operators to structure grammar specifications are the alternative '|' and Kleene's '$*$' for repetitions:

$$X \mid Y \to X \qquad X \mid Y \to Y \qquad star\ X \to \text{""} \qquad star\ X \to X{+}{+}star(X)$$

With them *letter* and *word* could be redefined as follows:

$$letter \to \text{"a"} \mid \text{"b"} \mid ... \mid \text{"z"} \qquad word \to star(letter)$$

The annoying fact is that this does not work! At least not under call-time choice, which implies that this is an uncorrect definition of *star* in systems like *Curry* or *Toy*. The problem with call-time choice here is that all the occurrences of *letter* created by *star* will be shared and therefore *word* will only generate words like *aaa*, *nnnn*, ..., made with repetitions of the same letter. To overcome this problem, we would like that in the definition of *word*, the application of

the *star* operation to the string generator *letter* could follow a run-time choice regime, so that each of the two occurrences of *letter* created in the rewriting steps

$$word \rightarrow star(letter) \rightarrow letter \mathbin{++} star(letter)$$

could evolve independently. In our proposed extension this would be expressed by writing the definition of *word* as follows:

$$word \rightarrow star(rt(letter))$$

where rt is a special unary function symbol indicating that its argument (*letter* in this case) is not going to be shared in the evaluation of the surrounding application (*star(rt(letter))* in this case).

We remark that in this example neither call-time nor run-time choice are a good single option as semantics for the whole program. The definition of *palindrome* requires call-time choice, while the use of *star* in *word* requires run-time choice. To the best of our knowledge, no existing implementation for functional logic programming offers the possibility of combining in the same program both kind of semantics. This paper addresses that problem at a practical level, aiming at a solution that can be easily realized by modifying existing Prolog-based functional logic systems. Although our main interest is easiness of implementation, we provide also formal calculi attempting to reflect at an abstract level the operational behavior of the extended language. These calculi could be the technical basis for a thorough investigation of the formal properties of our proposal, a matter that is out of the scope of this paper.

2 A tiny functional logic language with run-time choice annotations

We shortly present here a functional logic language with run-time choice annotations. To keep the presentation simple, we consider only a first order untyped core with the usual first order syntax of term rewriting systems. However, the implementation described in Sect. 5 extends the existing system *Toy*, which is a HO typed system using curried notation.

We consider a signature Σ made of constructor symbols $c, d, \ldots \in CS$, function symbols $f, g, \ldots \in FS$, the special unary symbol rt, and a set of variables $X, Y, \ldots \in \mathcal{V}$. We sometimes write $c \in CS^n$ ($f \in FS^n$) to denote a constructor (function) symbol of arity n. Constructor terms (or c-terms) $t, s, \ldots \in CTerm$ follow the syntax: $t ::= X \mid c(t_1, \ldots, t_n)$, and expressions (with run-time choice annotations) $e, \ldots \in RtExpr$ follow the syntax: $e ::= X \mid c(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n) \mid rt(e)$. An intermediate set between $CTerm$ and $RtExpr$ is the set $RtCTerm$ of annotated c-terms $RtCTerm \ni t ::= X \mid c(t_1, \ldots, t_n) \mid rt(e)$, where $t_1, \ldots, t_n$ are also from $RtCTerm$ and e is any expression.

A *program* is a set of function defining rules, each of the form

$$f(t_1, \ldots, t_n) \rightarrow e$$

where $(t_1, \ldots, t_n)$ is a linear tuple of c-terms from $CTerm$, and e is any expression from $RtExpr$. We remark that annotated c-terms play no special role in the syntax of programs, but play an important role in the parameter passing mechanism, which informally can be explained as follows: to apply a program rule $f(t_1, \ldots, t_n) \to e$ to a function application $f(e_1, \ldots, e_n)$, a matching substitution θ such that $f(t_1, \ldots, t_n)\theta \equiv f(e_1, \ldots, e_n)$ must exist, and then $f(e_1, \ldots, e_n)$ reduces to $r\theta$, but following the informal criterion about *sharing*: the copies of subexpressions e of $f(e_1, \ldots, e_n)$ created in $r\theta$ are not shared –i.e. follow run-time choice– if e is under a rt annotation, and shared –i.e. follow call-time choice– otherwise. These ideas are formalized in the next section in the form of two alternative operational calculi.

3 Formal operational calculi

In this section we will try to design some calculi able to express an extension of the standard call-time choice semantics for FLP [4], to support the primitive rt for run-time choice evaluation. Our approach to formalize this extension is based in two main ideas:

- The new calculus will be a modification of the simple rewrite calculus presented in [9]. As we will have to express run-time evaluation for parts of the computation, we will need to have partially evaluated expressions at our disposal. A calculus in the line of those used in [4] would not be a suitable tool, as it returns only partial values for the expressions, but no intermediate states of the computation.
- Instead of giving a semantics for annotations $rt(e)$ directly, we will think about it as a syntactic sugar for the annotation of the function symbols that appear in e with a rt superscript, indicating that those function symbols will be treated as a constructor symbol as far sharing and parameter passing is concerned. Therefore, an expression containing only variables, constructor symbols and function symbols annotated with rt could be copied freely, thus getting a run-time behaviour for it, as a function argument. We write FS^{rt} for the set of function symbols with superscript rt, $FS^{?}$ for $FS \cup FS^{rt}$ and $f^{?}$ for function symbols in $FS^{?}$, i.e., for possibly superscripted function symbols.

The desugaring of expressions to eliminate the rt primitive transforming it into rt annotations is performed according to the following definition:

Definition 1 (Desugaring of the rt primitive).

$$
\begin{array}{lll}
desugar(rt(X)) & = X & \text{if } X \in \mathcal{V} \\
desugar(rt(c(e_1, \ldots, e_n))) & = c(desugar(rt(e_1)), \ldots, desugar(rt(e_n))) & \text{if } c \in CS \\
desugar(rt(f(e_1, \ldots, e_n))) & = f^{rt}(desugar(rt(e_1)), \ldots, desugar(rt(e_n))) & \text{if } f \in FS \\
desugar(rt(rt(e))) & = desugar(rt(e)) &
\end{array}
$$

According to this syntactic desugaring for $rt(e)$, the syntax of annotated c-terms and expressions can be reformulated as follows:

- $RtCTerm \ni t ::= X \mid c(t_1,\ldots,t_n) \mid f^{rt}(t_1,\ldots,t_n)$, if $X \in \mathcal{V}$, $c \in CS^n$, $f \in FS^n$, $t_1,\ldots,t_n \in RtCTerm$
- $RtExpr \ni e ::= X \mid c(e_1,\ldots,e_n) \mid f^{?}(e_1,\ldots,e_n)$, if $X \in \mathcal{V}$, $c \in CS^n$, $f^{?} \in FS^{?}$, $e_1,\ldots,e_n \in RtExpr$

To express parameter passing in function applications with $rt-$annotated arguments we will need to consider rt-c-substitutions, defined by: $\theta \in RtCSubst$ iff $X\theta \in RtCTerm, \forall X \in \mathcal{V}$.

Now we will define calculi to work with annotated expressions. In [9] two rewrite notions for call-time choice were defined, each of them being interesting for different applications. Here we will modify both of them to get two (hopefully) equivalent characterizations of a semantics for annotated run-time choice under a call-time choice environment.

(B)	$\mathcal{C}[e] \rightarrowtail \mathcal{C}[\bot]$	for any context $\mathcal{C}$ and expression $e \in RtExpr_\bot$
(OR)	$\mathcal{C}[f^{?}(\overline{p})\theta] \rightarrowtail \mathcal{C}[r\theta]$	for any context $\mathcal{C}$, $(f(\overline{p}) \to r \in)\mathcal{P}$, and $\theta \in RtCSubst_\bot$

Fig. 1. A one-step reduction relation for non-strict call-time choice with rt annotations

The first characterization is shown in Fig. 1. Its drawback is that the rule **(B)** involves a 'magical' guessing in advance of the fact that the reduction of a (sub)-expression is not going to be needed, and replaces this 'no need of reduction in the future' by an artificial anticipated reduction to the undefined value $\bot$. However, because of its simplicity, the relation is helpful to understand what are the possible results of a reduction.

The second characterization is the rewrite relation of Fig. 2. It expresses in a more realistic manner (specially, if a reduction strategy would be added, which is not our focus here) the way in which computations are to be performed. To express sharing (when needed), local bindings are created via a *let* construct.

(Fapp) $f^{?}(\overline{p})\theta \ \to_l \ r\theta$, if $(f(\overline{p}) \to r) \in \mathcal{P}$, $\theta \in RtCSubst$

(LetIn) $h(\ldots,e,\ldots) \to_l let\ X = e\ in\ h(\ldots,X,\ldots)$, if $h \in \Sigma$, $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable

(Bind) $let\ X = t\ in\ e \ \to_l \ e[X/t]$, if $t \in RtCTerm$

(Elim) $let\ X = e_1\ in\ e_2 \to_l e_2$, if $X \notin FV(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \ \to_l \ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \to_l e'$ using any of the previous rules, and in case $e \to_l e'$ is a (Fapp) step using $(f(\overline{p}) \to r)\theta \in [\mathcal{P}]$ then $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$.

Fig. 2. Rules of *let*-rewriting extended with rt annotations

Note how in the rule (LetIn), in the case a function application is extracted to a *let*, it is needed that f is not marked with rt, which tell us that it is not allowed to duplicate it, and therefore it may be needed to put it in a *let* in order to progress with the evaluation (for example if it appears in an argument of another function application whose reduction is needed).

Example 1. Given the program

$$\begin{array}{ll} coin \to 0 & f(X) \to g(X, coin) \\ coin \to 1 & g(X, Y) \to (X, X, Y, Y) \end{array}$$

we want to evaluate the expression $rt(f(coin))$, which is desugared as $f^{rt}(coin^{rt})$. With the calculus of Fig. 1 we can do:

$$\begin{array}{l} f^{rt}(coin^{rt}) \rightarrowtail g(coin^{rt}, coin) \rightarrowtail g(coin^{rt}, 0) \rightarrowtail (coin^{rt}, coin^{rt}, 0, 0) \\ \rightarrowtail (0, coin^{rt}, 0, 0) \rightarrowtail (0, 1, 0, 0) \end{array}$$

Note how in the first step the expression $f^{rt}(coin^{rt})$ can be evaluated as every function symbol present in $coin^{rt}$ is annotated with rt. On the other hand we cannot apply (OR) to $g(coin^{rt}, coin)$, as one of its arguments contains a function symbol that it is not annoted for run-time, and thus the value $(0, 1, 0, 1)$ is not reachable from $f^{rt}(coin^{rt})$. This is even more evident in the version of this evaluation got with the calculus of Fig. 2:

$$\begin{array}{l} f^{rt}(coin^{rt}) \to_l g(coin^{rt}, coin) \to_l let\ X = coin\ in\ g(coin^{rt}, X) \\ \to_l let\ X = coin\ in\ (coin^{rt}, coin^{rt}, X, X) \to_l let\ X = coin\ in\ (0, coin^{rt}, X, X) \\ \to_l let\ X = coin\ in\ (0, 1, X, X) \to_l let\ X = 0\ in\ (0, 1, X, X) \\ \to_l (0, 1, 0, 0) \end{array}$$

When we reach the expression $let\ X = coin\ in\ (coin^{rt}, coin^{rt}, X, X)$ it is clear that the first two components of the tuple may evolve in different ways while the values of the last two components will be shared.

4 A variant of run-time annotations

In the present section we will show another primitive to express run-time choice that we will build on top of the previous primitive rt, through a simple program transformation. We will call that primitive rRt, and define its behaviour by the following inference rule that should be added to the *CRWL* logic [4]:

$$\frac{e \to^*_{\mathcal{P}'} e' \quad t \sqsubseteq |e'|}{\mathcal{P} \vdash_{CRWL} rRt(e) \twoheadrightarrow t} \quad (\mathbf{rRt})$$

where $\mathcal{P}'$ is the program resulting of adding to $\mathcal{P}$ the new rule $rRt(e) \to e$, and $e \to^*_{\mathcal{P}'} e'$ indicates that e' can be reached from e by zero or more steps of ordinary rewriting [2] using the program $\mathcal{P}'$. The approximation ordering $t \sqsubseteq t'$ between partial values expresses that t is less defined than t' (see [4] for details).

The rule (**rRt**) itself is already suggesting a possible implementation for rRt. This implementation will be based on the fact that, for any program in which every function symbol that appears in a right hand side of a program rule is rt-annotated, the evaluation of an expression that has each of its function symbols rt-annotated too returns the same results as it was evaluated under run-time choice but discarding the annotations. This ideas are formalized in the following definition:

Definition 2. *Given a CRWL-program $\mathcal{P}$:*

- *We build the signature of a new program $\_\mathcal{P}$ adding to it any constructor symbol in the signature of $\mathcal{P}$, and for any function symbol f in the signature of $\mathcal{P}$ considering a fresh function symbol $\_f$ which we add to the signature of $\_\mathcal{P}$.*
- *We define the transformation of expressions rRt as:*

$$\begin{array}{lll} rRtT(X) & = X & \text{if } X \in \mathcal{V} \\ rRtT(c(e_1,\ldots,e_n)) & = c(rRtT(e_1),\ldots,rRtT(e_n)) & \text{if } c \in CS \\ rRtT(f(e_1,\ldots,e_n)) & = \_f^{rt}(rRtT(e_1),\ldots,rRtT(e_n)) & \text{if } f \in FS \\ rRtT(rRtT(e)) & = rRtT(e) & \end{array}$$

- *For any $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ we add the rule $\_f(p_1,\ldots,p_n) \to rRtT(r)$ to $\_\mathcal{P}$.*

Finally, any expression $rRt(e)$ to be evaluated under $\mathcal{P}$ is desugared into $rRtT(e)$ and evaluated under $\mathcal{P} \uplus \_\mathcal{P}$

Example 2. Starting with the program of Example 1 we get the program

$$\begin{array}{c} \{coin \to 0, coin \to 1, f(X) \to g(X, coin), g(X,Y) \to (X,X,Y,Y)\} \\ \uplus \\ \{\_coin \to 0, \_coin \to 1, \_f(X) \to \_g^{rt}(X, coin), \_g(X,Y) \to (X,X,Y,Y)\} \end{array}$$

under which we can do:

$$\begin{array}{l} rRt(f(coin)) \equiv \_f^{rt}(\_coin^{rt}) \rightarrowtail \_g^{rt}(\_coin^{rt}, \_coin^{rt}) \\ \rightarrowtail (\_coin^{rt}, \_coin^{rt}, \_coin^{rt}, \_coin^{rt}) \rightarrowtail^* (0,1,0,1) \end{array}$$

5 Implementation issues

In order to study the practicability of the proposal we have implemented it as an extension of the functional logic system *Toy* ([3]). This system, as well as other modern systems like *Curry* ([6]), operates under call-time choice. We introduce the new syntactic construct *rt e* into the syntax of *Toy* to instruct the system to evaluate the expression e under a run-time choice regime. The system will use run-time choice for evaluating the expressions annotated with *rt*, and call-time choice as usual for the rest of computations, i.e., we have within the same language both regimes of evaluation.

The extension is well supported by the system and requires only some lightweight modifications. In fact, the traditional problem is how to achieve sharing in a non-deterministic language like this, and our goal now is to inhibit this sharing mechanism at the points required by the programmer with *rt*.

Toy is implemented in Prolog and uses Prolog as target code (see [8, 3] for details). Sharing is implemented by means of *suspensions*, that are Prolog terms of the form:

susp(FunctionName,Arguments,Result,Evaluated)

The *FunctionName* and its *Arguments* represent the expression e to be evaluated, while *Result* is the resulting value (if evaluated, variable in other case) and *Evaluated* is a flag that indicates if the expression has been evaluated (flag *on*) or not (flag variable). Every function call is translated into a suspension in order to share its value when the expression is passed as argument and copied. As an example of the use of this representation consider the following program:

```
coin = 0
coin = 1

double X = X + X

test1 = double coin
test2 = rt (double coin)
```

Consider the evaluation of *test1*. As all the function calls are translated into suspended forms, in particular *coin* will have the form *susp(coin,[],R,E)*. The evaluation of *double* does not demand the evaluation of its argument *coin*, so it will produce

susp(coin,[],R,E) + susp(coin,[],R,E)

Later, when one of the calls to *coin* is evaluated, for example to 0, the other one automatically gets the same value:

susp(coin,[],0,on) + susp(coin,[],0,on)

The result of the addition is 0, that is a value obtained for *test1*. If we evaluate *coin* to 1 we have

susp(coin,[],1,on) + susp(coin,[],1,on)

and then result 2, that is the other value obtained for *test1*. With this sharing mechanism we can not obtain the value 1 for *double coin* as it would require to evaluate both calls to *coin* to two different values.

For the function function *test2* we would want to obtain the values 0 and 2 as before, but also the value 1 (evaluating separately both calls to *coin*). In this case *rt* will deactivate the sharing mechanism. This can be easily achieved by translating the call *coin* into the suspended form *susp(coin,[],R,rt)*. The flag *rt* will indicate to the system that the value of this expression must not be shared (and neither kept in the variable R). For *test2* we evaluate

susp(coin,[],R,rt) + susp(coin,[],R,rt)

The first suspension can be reduced to 0 (without annotating the result in R), and the second one to 1, obtaining 1 for *test2* as expected.

The extension implemented in *Toy* provides this behaviour with *test1* and *test2*. In fact, for *test2* it obtains 0, 2 and 1 twice (evaluating the first *coin* to 0 and the second to 1 and viceversa). As another example, consider the problem of generating numbers as combinations of the digits 0, 1 and 2. Using *take*, *repeat* and the alternative operator '|' (introduced in Sec. 1) we could define:

```
number N = take N (repeat (0 | 1 | 2))
```

but then the expression *number 3* will produce only the answers [0,0,0], [1,1,1] and [2,2,2], because the expression *0 | 1 | 2* is evaluated only once and then its value is shared when evaluating *repeat*. For achieving the expected behaviour we have to instruct the system for choosing the digits under run-time choice (to avoid sharing):

```
number N = take N (repeat (rt (0 | 1 | 2)))
```

Now we obtain the 27 possible combinations that include $[1, 1, 2]$ or $[3, 1, 2]$ as instance. The example of palindromes of Sect. 1 also works as expected.

The prototype and some examples can be found at
`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`.

6 Conclusions

We have proposed a simple way of combining in the same program run-time choice and call-time choice, two semantics commonly adopted for non-determinism in rewriting-based declarative languages, but that cannot coexist within the same program in current systems.

The approach presented here starts from a call-time choice ambient (as given by most popular functional logic systems like *Curry* [6] or *Toy* [11]) and adds to it the possibility of annotating the evaluation of (sub)-expressions as following a run-time choice regime. We have proposed two variants of this idea, the first being more 'local' in the effect of an annotation $rt(e)$, while the second is more global. In both cases we have proposed a formal definition of the intended semantics.

For the first variant we have given formal operational descriptions, by adapting to the new setting two one-step reduction relations proposed in [9] as a simple notion of rewriting adequate for call-time choice. As for implementation, this variant has been achieved by modifying of the system *Toy*. Essentially, we have needed to change the management of *suspensions*, that are the technical key to implement sharing for call-time choice. The resulting prototype can be found at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems`.

For the second variant we give a logical semantics that extends, to cope with *rt* annotations, the proof calculus of the *CRWL* framework [4]. We have seen

how to transform annotations of this variant into the first one. This mapping can be used to implement the second variant.

Recently, we have tried a different alternative to the combination of call-time and run-time choice [10], following a way complementary to the one in this paper: there we start from ordinary rewriting and enhance it with local bindings through a *let* construct to express sharing and call-time choice. The resulting framework seems to be more amenable to formal treatments, as shown by the good number of technical results obtained in [10]. On the other hand, the approach here seems to be more easily implementable, at least if one wants to reuse existing call-time-choice based implementations.

References

1. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998.
3. R. Caballero and J. Sánchez (eds.). TOY: A multiparadigm declarative language, version 2.2.3. Technical report, UCM, Madrid, July 2006.
4. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
5. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
6. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
7. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
8. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.
9. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming*, pages 197–208. ACM Press, 2007.
10. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions (extended version). Technical report, 2008. http://gpd.sip.ucm.es/juanrh/pubs/tchrRTCT08.pdf.
11. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
12. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

7.2.5 The Full Abstraction Problem for Higher Order Functional-Logic Programs

The Full Abstraction Problem for Higher Order Functional-Logic Programs

F.J. López-Fraguas and J. Rodríguez-Hortalá *

Departamento de Sistemas Informáticos y Computación
fraguas@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. Developing suitable formal semantics can be of great help in the understanding, design and implementation of a programming language, and act as a guide for software development tools like analyzers or partial evaluators. In this sense, full abstraction is a highly desirable property, indicating a perfect correspondence between the semantics and the observable behavior of program pieces. In this work we address the question of full abstraction for the family of modern functional logic languages, in which functions can be higher order and non-deterministic, and where the semantics adopted for non-determinism is *call-time choice*. We show that, with respect to natural notions of *observation*, any semantics based on *extensional* functions is necessarily unsound; in contrast, we show that the higher order version of *CRWL*, a well-known existing semantic framework for functional logic programming, based on an *intensional* view of functions, turns out to be fully abstract and compositional.

1 Introduction

Developing suitable formal semantics can be of great help in the understanding, design and implementation of a programming language, and acts as a guide for software development tools like analyzers or partial evaluators. In this sense, *full abstraction* is a highly desirable property, indicating a perfect correspondence between the semantics and the behavior of program pieces, according to a given criterion of *observation*.

The notion of full abstraction was introduced by Plotkin [19] in connection to PCF, a simple model of functional programming based on λ-calculus. He realized that the standard Scott semantics, in which expressions of functional types have classical mathematical functions as meanings, lacks full abstraction with respect to observing the value obtained in the evaluation of an expression. The reason lays in the impossibility of defining the function *por* (*parallel or*) in PCF. Using this fact one can build two higher order (HO) expressions e_1, e_2 denoting two different mathematical functions φ_1, φ_2, both expecting boolean functions as arguments, such that φ_1, φ_2 only differ when applied to *por* as

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03, TIN2008-06622-C03-01, S-0505/TIC/0407 and UCM-BSCH-GR58/08-910502.

argument. Therefore e_1, e_2 have different Scott semantics but this difference cannot be *observed*. It is usually said that the semantics is *too concrete*. Notice, however, that Scott semantics for PCF is *sound*, that is, if two expressions have the same semantics, they cannot be observably distinguished. Unsoundness of a semantics can be considered a flaw, much more severe that being too concrete, which is more a weakness than a flaw.

Full abstraction for PCF was achieved in different technical ways (see e.g. [3]). But for our purposes it is more interesting to recall that the Scott semantics becomes fully abstract if PCF is enriched with the 'missing' *por* function (see e.g. [18]). The mainstream of functional logic programming (FLP, see [10]) is based rather in the theory of term rewriting systems than in λ-calculus; a consequence is that parallel or can be defined straightforwardly by an overlapping (almost orthogonal) rewriting system. So one could think of assigning to FLP languages a denotational semantics in the FP style. For instance, for a definition like $f\ x = 0$, one could assign to f the meaning $\lambda x.0$. The next step of our discussion is taking into account that modern FLP languages like Curry [12] or Toy [16] also permit non-confluent and non-terminating programs that define non-deterministic non-strict functions. This suggests that the standard semantics should be modified in the sense that the meaning of a function would be some kind of set-valued function.

The starting motivation of this paper is that *this roadmap cannot be followed anymore when non-determinism is combined with HO*, at least when considering *call-time choice* [13, 9], which is the notion of non-determinism adopted in, e.g., Curry or Toy. The following example taken from [15] shows it:

Example 1. The following program computes with natural numbers represented by the constructors 0 and s, and where + is defined as usual. The syntax uses HO curried notation.

```
g X -> 0          f -> g          f' X -> f X
h X -> s 0        f -> h

fadd F G X -> (F X) + (G X)       fdouble F -> fadd F F
```

Here f and f' are non-deterministic functions that are (by definition of f') extensionally equivalent. In a set-valued variant of Scott semantics, their common denotation would be the function $\lambda X.\{0, s\ 0\}$, or something essentially equivalent. But this leads to unsoundness of the semantics. To see why, consider the expressions *(fdouble f 0)* and *(fdouble f' 0)*. In Curry or Toy, the possible values for *(fdouble f 0)* are *0, s (s 0)*, while *(fdouble f' 0)* can be in addition reduced to *s 0*. The operational reason to this situation is that *fdouble f 0* is rewritten first to *fadd f f 0* and then to *f 0 + f 0*; now, call-time choice enforces that evaluation of the two created copies of f (which is an evaluable expression) must be shared. In the case of *f' 0 + f' 0*, since f' is a normal form, the two occurrences of *f' 0* evolve independently. We see then that f and f' can be put in a context able to distinguish them, implying that any semantics assigning f and f' the same denotation is necessarily unsound, and therefore not fully abstract.

The combination *HO + Non-determinism + call-time choice* was addressed in *HOCRWL* [7, 8], an extension to HO of *CRWL* [9], a semantic FO framework specifically devised for FLP with call-time choice semantics for non-determinism. *HOCRWL* adopts an *intensional* view of functions, where different descriptions – in the form of *HO-patterns*– of the same extensional function are distinguished as different data. The intensional point of view of *HOCRWL* was an *a priori* design decision, motivated by the desire of achieving enough power for HO programming while avoiding the complexity of higher-order unification of λ-terms modulo $\beta\eta$, followed in other approaches [17, 11]. The issues of soundness or full abstraction were not the (explicit nor implicit) concerns of [7, 8]; whether *HOCRWL* actually fulfils those properties or not is exactly the question considered in this paper. As we will get positive answers, an anticipated conclusion of our work is that one must take into account intensional descriptions of functions as sensible meanings of expressions in HO non-deterministic FLP programs, even if one does not want to explicitly program with HO-patterns.

The rest of the paper is organized as follows. Next section recalls some essential preliminaries about applicative HO rewrite systems and the *HOCRWL* framework. We introduce also some terminology about semantics and extensionality needed for Sect. 3, where we examine soundness and full abstraction with respect to reasonable notions of observation based on the result of reductions. The section ends with a discussion of the problems encountered when programs have *extra* variables, i.e., variables occuring in right, but not in left-hand sides of function defining rules. Finally Sect. 4 summarizes some conclusions and future work.

2 Higher-Order Functional-Logic Programs

2.1 Expressions, patterns and programs

We consider *function* symbols $f, g, \ldots \in FS$, *constructor* symbols $c, d, \ldots \in CS$, and *variables* $X, Y, \ldots \in \mathcal{V}$; each $h \in FS \cup CS$ has an associated *arity*, $ar(h) \in \mathbb{N}$; FS^n (resp. CS^n) is the set of function (resp. constructor) symbols with arity n. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects o. The set of *applicative expressions* is defined by $Exp \ni e ::= X \mid h \mid (e_1\ e_2)$. As usual, application is left associative and outer parentheses can be omitted, so that $e_1\ e_2\ \ldots e_n$ stands for $((\ldots(e_1\ e_2)\ldots)\ e_n)$. The set of variables occurring in e is written by $var(e)$. A distinguished set of expressions is that of *patterns* $t, s \in Pat$, defined by: $t ::= X \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_m$, where $0 \leq n \leq ar(c), 0 \leq m < ar(f)$. Patterns are irreducible expressions playing the role of *values*. *FO-patterns*, defined by $FOPat \ni t ::= X \mid c\ t_1 \ldots t_n\ (n = ar(c))$, correspond to FO constructor terms, representing ordinary non-functional data-values. Partial applications of symbols $h \in FS \cup CS$ to other patterns are HO-patterns and can be seen as truly data-values representing functions from an *intensional* point of view. Examples of patterns with the signature of Ex. 1 are: *0, s X, s, f', fadd f' f'*. The last three are HO-patterns. Notice that *f, fadd f f* are not patterns since f is not a pattern ($ar(f) = 0$).

4

Contexts are expressions with a hole defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$. Application of $\mathcal{C}$ to e (written $\mathcal{C}[e]$) is defined by $[\,][e] = e$; $(\mathcal{C}\ e')[e] = \mathcal{C}[e]\ e'$; $(e'\ \mathcal{C})[e] = e'\ \mathcal{C}[e]$. Substitutions $\theta \in Subst$ are finite mappings from variables to expressions; $[X_i/e_i, \ldots, X_n/e_n]$ is the substitution which assigns $e_i \in Exp$ to the corresponding $X_i \in \mathcal{V}$. We will mostly use *pattern-substitutions* (or simply *p-substitutions*) $PSubst = \{\theta \in Subst \mid \theta(X) \in Pat, \forall X \in \mathcal{V}\}$.

As usual while describing semantics of non-strict languages, we enlarge the signature with a new 0-ary constructor symbol $\bot$, which can be used to build the sets $Expr_\bot, Pat_\bot, PSubst_\bot$ of *partial* expressions, patterns and p-substitutions resp.

A *HOCRWL*-program (or simply a *program*) consists of one or more *program rules* of the form $f\ t_1 \ldots t_n \to r$ where $f \in FS^n$, $(t_1, \ldots, t_n)$ is a linear (i.e. variables occur only once) tuple of (maybe HO) patterns and r is any expression. Notice that confluence or termination is not required. In the present work we restrict ourselves to programs not containing *extra variables*, i.e., programs for which $var(r) \subseteq var(f\ \overline{t})$ holds for any program rule. There are technical reasons for such limitation (see Sect. 3.2), whose practical impact is on the other hand mitigated by known extra-variables elimination techniques [4, 2]. *HOCRWL*-programs often allow also *conditions* in the program rules. However, programs with conditions can be transformed into equivalent programs without conditions; therefore we consider only unconditional rules.

Some FLP systems, like Curry, do not allow HO-patterns in left-hand sides of function definitions. We call *left-FO* programs to these special kind of *HOCRWL*-programs. We remark that all the notions and results in the paper are applicable to *left-FO* programs and we stress the fact that Ex. 1 is one of them.

2.2 The *HOCRWL* proof calculus [7]

The semantics of a program $\mathcal{P}$ is determined in *HOCRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in Pat_\bot$, meaning informally that t is (or approximates to) a possible value of e, obtained by evaluation of e using $\mathcal{P}$ under call-time choice.

The *HOCRWL*-proof calculus is presented in Fig. 1. We write $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in that calculus using the program $\mathcal{P}$. The *HOCRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}}_{HOCRWL} = \{t \in Pat_\bot \mid \mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t\}$. $\mathcal{P}$ and *HOCRWL* are frequently omitted in those notations.

Looking at in Ex. 1 we have $[\![f double\ f\ 0]\!] = \{0, s\ (s\ 0), \bot, s\ \bot, s\ (s\ \bot)\}$ and $[\![f double\ f'\ 0]\!] = \{0, s\ 0, s\ (s\ 0), \bot, s\ \bot, s\ (s\ \bot)\}$.

We will use the following result stating an important compositionality property of the semantics of *HOCRWL*-expressions: the semantics of a whole expression depends only on the semantics of its constituents, in a particular form reflecting the idea of call-time choice.

Theorem 1 (Compositionality of *HOCRWL* semantics, [15]). *For any* $e \in Exp_\bot$, $\mathcal{C} \in Cntxt$, $[\![\mathcal{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathcal{C}[t]]\!]$.

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(RR)}\ \frac{}{x \twoheadrightarrow x} \qquad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1 \ \ldots\ e_n \twoheadrightarrow t_m}{h\ e_1 \ldots e_m \twoheadrightarrow h\ t_1 \ldots t_m} \qquad h \in \Sigma,\ \text{if } h\ t_1 \ldots t_m \text{ is a partial pattern, } m \geq 0$$

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow p_1\theta \ldots\ e_n \twoheadrightarrow p_n\theta \quad r\theta\ a_1 \ldots a_m \twoheadrightarrow t}{f\ e_1 \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t} \qquad \begin{array}{l}\text{if } m \geq 0, \theta \in PSubst_\bot \\ (f\ p_1 \ldots p_n \to r) \in \mathcal{P}\end{array}$$

Fig. 1. (*HOCRWL*-calculus)

The *HOCRWL* logic is related to several operational notions. In [7] a goal solving narrowing calculus was presented and its strong adecuacy to *HOCRWL* shown. The operational semantics of [1] has been also used in many works in the field of FLP. Its equivalence with the first order version of *HOCRWL* was stated in [14], and it can be transfered to higher order through the results of [15, 1]. The formalization of graph rewriting of [5, 6] has been often used in FLP too, and although never formally proved, it is usually considered that it specifies the same behaviour. Finally, in [15] a notion of higher order rewriting with local bindings called *HOlet-rewriting* and its lifting to narrowing were proposed, and its adequacy to *HOCRWL* was formally proved. It can be summarized in the following result:

Theorem 2 ([15]). *$\forall e \in Exp,\ t \in Pat,\ t \in [\![e]\!]^{\mathcal{P}}$ iff $\mathcal{P} \vdash e \to^{l^*} t$, where $\to^{l^*}$ stands for the reflexive-transitive closure of the HOlet-rewriting relation.*

Therefore, we can use the set of total values computed for an expression in *HOCRWL* as a characterization of the operational behaviour of that expression, as it has a strong correspondence, not only with its behaviour under *HOlet*-rewriting, but also under any of the operational notions metioned above.

2.3 Extensionality

In order to achieve more generality and technical precision wrt. the discussion of Ex.1, we introduce here some new terminologies and notations about extensional equivalence and related notions that will be used later on. They can be expressed in terms of the *HOCRWL* semantics $[\![\_]\!]$.

Definition 1 (Extensional equivalence, extensional semantics).

(i) Given $n \geq 0$, two expressions $e, e' \in Expr_\bot$ are said to be n-extensionally equivalent *($e \sim_n e'$) iff $[\![e\ e_1 \ldots e_n]\!] = [\![e'\ e_1 \ldots e_n]\!]$, for any $e_1, \ldots, e_n \in Expr_\bot$.*

(ii) Given $n \geq 0$, $e \in Expr_\bot$, the n-extensional semantics of e is defined as: $[\![e]\!]_{ext_n} = \lambda t_1 \ldots \lambda t_n.\ [\![e\ t_1 \ldots t_n]\!]$ ($t_i \in Pat_\bot$).

We can establish some relationships between these notions:

6

Proposition 1.

(i) $e \sim_n e' \Rightarrow e \sim_m e'$, *for all* $m > n$.
(ii) $e \sim_n e' \Leftrightarrow [\![e\ t_1 \dots t_n]\!] = [\![e'\ t_1 \dots t_n]\!]$, *for any* $t_1, \dots, t_n \in Pat_\bot$.
(iii) $e \sim_n e' \Leftrightarrow [\![e]\!]_{ext_n} = [\![e']\!]_{ext_n}$

Proof. The proof is easy, thanks to compositionality of $[\![\_]\!]$ (Th. 1).

(i) Assume $e \sim_n e'$, $m > n$, let $e_1 \dots e_m \in Expr_\bot$. We must prove $[\![e\ e_1 \dots e_m]\!] = [\![e'\ e_1 \dots e_m]\!]$. We reason as follows:

$$\begin{array}{ll} [\![e\ e_1 \dots e_m]\!] & = \\ [\![(e\ e_1 \dots e_n)e_{n+1} \dots e_m]\!] & = \text{(by compositionality)} \\ \bigcup_{t \in [\![e\ e_1 \dots e_n]\!]} [\![t\ e_{n+1} \dots e_m]\!] & = \text{(since } e \sim_n e') \\ \bigcup_{t \in [\![e'\ e_1 \dots e_n]\!]} [\![t\ e_{n+1} \dots e_m]\!] & = \text{(by compositionality)} \\ [\![(e'\ e_1 \dots e_n)e_{n+1} \dots e_m]\!] & = \\ [\![e'\ e_1 \dots e_m]\!] & \end{array}$$

(ii) Another direct use of compositionality
(iii) Consequence of *(i),(ii)* and definitions of $\sim_n, [\![\_]\!]_{ext_n}$.

3 CRWL and Full Abstraction

3.1 Full Abstraction

In this section we examine technically soundness and full abstraction of the *HOCRWL* semantics $[\![\_]\!]$ and its extensional variants $[\![\_]\!]_{ext_k}$. We can anticipate a positive answer for $[\![\_]\!]$ and negative for the others.

Full abstraction depends on a criterion of observability for expressions. In constructor based languages, like FLP languages, it is reasonable to observe the outcomes of computations, given by constructor forms reached by reduction. Here, we can interpret 'constructor form' in a liberal sense, including HO-patterns, or in a more restricted sense, only with FO-patterns. This leads to the following notions of observation.

Definition 2 (observations). *Let* $\mathcal{P}$ *be a program. We consider the following observations:*

- $\mathcal{O}^{\mathcal{P}} : Expr \mapsto Pat$ *is defined as* $\mathcal{O}^{\mathcal{P}}(e) = \{t \in Pat \mid \mathcal{P} \vdash e \to^{l^*} t\}$
- $\mathcal{O}^{\mathcal{P}}_{fo} : Expr \mapsto FOPat$ *is defined as* $\mathcal{O}^{\mathcal{P}}_{fo}(e) = \{t \in FOPat \mid \mathcal{P} \vdash e \to^{l^*} t\} (= \mathcal{O}^{\mathcal{P}}(e) \cap FOPat)$

We remark that, due to the strong correspondence between reduction and semantics given by Th. 2, we also have $\mathcal{O}^{\mathcal{P}}(e) = [\![e]\!]^{\mathcal{P}} \cap Pat$, implying in particular $\mathcal{O}^{\mathcal{P}}(e) \subseteq [\![e]\!]^{\mathcal{P}}$ (and similar conditions hold for $\mathcal{O}_{fo}$).

Now we turn to the definition of full abstraction. In programming languages like PCF the condition for full abstraction is usually stated as:

$$[\![e]\!] = [\![e']\!] \Leftrightarrow \mathcal{O}(\mathcal{C}[e]) = \mathcal{O}(\mathcal{C}[e']), \text{for any context } \mathcal{C} \tag{1}$$

where $\mathcal{O}$ is the observation function of interest. Programs do not need to be mentioned, because programs and expressions can be identified by contemplating the evaluation of e under $\mathcal{P}$ as the evaluation of a big λ-expression or big *let*-expression embodying $\mathcal{P}$ and e. Contexts pose no problems either. In our case, since programs are kept different from expressions, some care must be taken. It might happen that $\mathcal{P}$ has not enough syntactical elements and rules to built interesting distinguishing contexts. For instance, if in Ex. 1 we drop the definition of $fdouble$, and we consider $\mathcal{O}_{fo}$ as observation, then we cannot built a context that distinguishes f from f'. This would imply that soundness or full abstraction would not be intrinsic to the semantics, but would greatly depend on the program. What we need is requiring the right part of (1) to hold for all contexts that might be obtained by extending $\mathcal{P}$ with new auxiliary functions. To be more precise, we say that $\mathcal{P}'$ is a *safe extension* of $(\mathcal{P}, e)$ if $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}''$, where $\mathcal{P}''$ does not include defining rules for any function symbol occurring in $\mathcal{P}$ or e. The following property of *HOCRWL* regarding safe extensions will be crucial for full abstraction. The property is subtler than it appears to be, as witnessed by the fact that it fails to hold if programs have extra variables, as discussed in Sect. 3.2.

Lemma 1. $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$ *when* $\mathcal{P}'$ *safely extends* (P, e).

Proof. As $\mathcal{P} \subseteq \mathcal{P}'$ then $[\![e]\!]^{\mathcal{P}} \subseteq [\![e]\!]^{\mathcal{P}'}$ trivially holds, as every *HOCRWL*-proof for $\mathcal{P} \vdash e \twoheadrightarrow t$ is also a proof for $\mathcal{P}' \vdash e \twoheadrightarrow t$.

On the other hand, to prove the inclusion $[\![e]\!]^{\mathcal{P}'} \subseteq [\![e]\!]^{\mathcal{P}}$ let us precisely formalize the notion of safe extension. For any program $\mathcal{P}$, we write $defs(\mathcal{P})$ for the set of function symbols defined in $\mathcal{P}$ (i.e., appearing at the root of some left-hand side of a program rule of $\mathcal{P}$); for any expression e, we write FS^e for the set of function symbols appearing in e; for any program $\mathcal{P}$ and rule $(l \to r) \in \mathcal{P}$ we define $FS^{(l \to r)} = FS^l \cup FS^r$ and $fs^{\mathcal{P}} = \bigcup_{(l \to r) \in \mathcal{P}} FS^{(l \to r)}$. Then $\mathcal{P}'$ is a safe extension of $(\mathcal{P}, e)$ iff $\mathcal{P}' = \mathcal{P} \uplus \mathcal{P}''$ such that $defs(\mathcal{P}'') \cap (FS^e \cup FS^{\mathcal{P}}) = \emptyset$.

Now we will see that for any proof for $\mathcal{P}' \vdash a \twoheadrightarrow s$ if $defs(\mathcal{P}'') \cap FS^a = \emptyset$ then $defs(\mathcal{P}'') \cap FS^s = \emptyset$ and for any premise $a' \twoheadrightarrow s'$ appearing in that proof we have $defs(\mathcal{P}'') \cap (FS^{a'} \cup FS^{s'}) = \emptyset$, by induction on the structure of $\mathcal{P}' \vdash a \twoheadrightarrow s$. Let us do a case distinction over the rule applied at the root. If it was B then the only statement is $a \twoheadrightarrow \bot$ for which the condition holds because $\bot \notin FS$. If it was RR then the only statement is $x \twoheadrightarrow x$, but $x \notin FS$. If it was DC then we apply the IH over each $e_i \twoheadrightarrow t_i$, because $defs(\mathcal{P}'') \cap FS^{(h\ e_1 \ldots e_m)} = \emptyset$ implies $defs(\mathcal{P}'') \cap FS^{e_i} = \emptyset$ for each e_i. All that is left is checking that $defs(\mathcal{P}'') \cap FS^{(h\ t_1 \ldots t_m)} = \emptyset$. But $defs(\mathcal{P}'') \cap FS^{t_i} = \emptyset$ for each t_i by IH, and $h \in FS^{(h\ e_1 \ldots e_m)} \cap defs(\mathcal{P}'') = \emptyset$ by hypothesis, so we are done. Finally, for OR we apply the IH to $e_i \twoheadrightarrow p_i\theta$ and its premises, as we did in DC. Besides $f \in FS^{(f\ e_1 \ldots e_n\ a_1 \ldots a_m)} \cap defs(\mathcal{P}'') = \emptyset$ by hypothesis, so $(f\ p_1 \ldots p_m \to r) \in \mathcal{P}$, hence $defs(\mathcal{P}'') \cap FS^{(f\ p_1 \ldots p_m \to r)} = \emptyset$, because $\mathcal{P}''$ is a safe extension. Combining both facts with the absence of extra variables in program rules we get $FS^{r\theta} \cap defs(\mathcal{P}'') = \emptyset$. But $FS^{(f\ e_1 \ldots e_n\ a_1 \ldots a_m)} \cap defs(\mathcal{P}'') = \emptyset$ by hypothesis, hence

8

$FS^{(r\theta\ a_1 \ldots a_m)} \cap defs(\mathcal{P}'') = \emptyset$, to which we can apply the IH to conclude the proof.

Finally, assuming a proof $\mathcal{P}' \vdash e \twoheadrightarrow t$ we may apply the property above because $defs(\mathcal{P}'') \cap FS^e = \emptyset$, as $\mathcal{P}''$ is a safe extension. Therefore $\mathcal{P}''$ was not used in that proof and so it is also a proof for $\mathcal{P} \vdash e \twoheadrightarrow t$, since $\mathcal{P}' = \mathcal{P} \uplus \mathcal{P}''$.

We can now define:

Definition 3 (Full abstraction).

(a) A semantics is fully abstract *wrt $\mathcal{O}$ iff for any $\mathcal{P}$ and $e, e' \in Expr$, the following two conditions are equivalent:*
(i) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ (ii) $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$ for any $\mathcal{P}'$ safely extending $(\mathcal{P}, e)$, $(\mathcal{P}, e')$ and any $\mathcal{C}$ built with the signature of $\mathcal{P}'$.

(b) A notion weaker than full abstraction is: a semantics is sound *wrt $\mathcal{O}$ iff the condition* (i) *above implies the condition* (ii).

For extensional semantics, our Ex. 1 (and obvious generalizations to arities $k > 1$) constitutes a proof of the following negative result:

Proposition 2. *For any $k > 0$, $[\![\_]\!]_{ext_k}$ is unsound wrt $\mathcal{O}, \mathcal{O}_{fo}$. This remains true even if programs are restricted to be left-FO.*

This contrast with the following:

Theorem 3 (Full abstraction). *$[\![\_]\!]$ is fully abstract wrt $\mathcal{O}$ and $\mathcal{O}_{fo}$.*

The proof for this theorem will be based on the compositionality of $[\![\_]\!]$ and the following result:

Lemma 2. *Let $\mathcal{P}$ be any program. Consider the transformation $\hat{\_} : Pat_\perp \to Pat$ defined by:*

$$\hat{X} = X \qquad \hat{\perp} = bot \qquad \widehat{h\ t_1 \ldots t_m} = h\ \hat{t_1}\ \ldots\ \hat{t_m}$$

where bot is a fresh constant constructor symbol. Consider also the program $\mathcal{P}' = \mathcal{P} \uplus \mathcal{P}_{g_t}$, where $\mathcal{P}_{g_t}$ consists of the following rules defining some fresh symbols $g_s \in FS$:

$$g_X\ U \to U \qquad g_\perp\ X \to bot$$
$$g_{(h\ t_1\ \ldots t_m)}(h\ X_1\ \ldots X_m) \to h\ (g_{t_1} X_1) \ldots (g_{t_m} X_m)$$

Then:

(i) $\mathcal{P}'$ is a safe extension of $(\mathcal{P}, e)$.
(ii) $t \in [\![e]\!]^{\mathcal{P}}$ iff $\hat{t} \in [\![g_t\ e]\!]^{\mathcal{P}'}$, for any $e \in Exp_\perp, t \in Pat_\perp$ built with the signature of $\mathcal{P}$.

Proof. It is clear that $\mathcal{P}'$ is a safe extension as it only defines new rules for fresh function symbols. The other equivalence holds by two simple inductions on the structure of t.

Proof (For Theorem 3). First of all we will prove the full abstraction wrt. $\mathcal{O}$. We will see that $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ iff for any safe extension $\mathcal{P}'$ of $(\mathcal{P}, e)$ and $(\mathcal{P}, e')$, for any context $\mathcal{C}$ built with the signature of $\mathcal{P}'$ we have $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$. Concerning the left to right implication, assume $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ and fix some safe extension $\mathcal{P}'$ and some context $\mathcal{C}$ built on it. First we will see that $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) \subseteq \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$. Assume some $t \in \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e])$, then $t \in [\![\mathcal{C}[e]]\!]^{\mathcal{P}'}$ by definition and Th. 2. But then

$$
\begin{array}{ll}
t \in [\![\mathcal{C}[e]]\!]^{\mathcal{P}'} = \bigcup_{t \in [\![e]\!]^{\mathcal{P}'}} [\![\mathcal{C}[t]]\!]^{\mathcal{P}'} & \text{by Th. 1} \\
= \bigcup_{t \in [\![e]\!]^{\mathcal{P}}} [\![\mathcal{C}[t]]\!]^{\mathcal{P}'} & \text{by Lemma 1, as } \mathcal{P}' \text{ is a safe extension} \\
= \bigcup_{t \in [\![e']\!]^{\mathcal{P}}} [\![\mathcal{C}[t]]\!]^{\mathcal{P}'} & \text{by hypothesis} \\
= \bigcup_{t \in [\![e']\!]^{\mathcal{P}'}} [\![\mathcal{C}[t]]\!]^{\mathcal{P}'} & \text{by Lemma 1, as } \mathcal{P}' \text{ is a safe extension} \\
= [\![\mathcal{C}[e']]\!]^{\mathcal{P}'} & \text{by Th. 1}
\end{array}
$$

But then $t \in \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$ by definition and Th. 2. The other inclusion can be proved in a similar way.

Regarding the right to left implication, we will use the transformation $\hat{\_}$ of Lemma 2. We can also take the program $\mathcal{P}'$ of Lemma 2 which is a safe extension of $(\mathcal{P}, e)$ and $(\mathcal{P}, e')$ as it only defines new rules for fresh function symbols. Therefore we can assume $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$ for any $\mathcal{C}$ built on $\mathcal{P}'$. Besides, for any $t \in [\![e]\!]^{\mathcal{P}}$ we have $\hat{t} \in [\![g_t\ e]\!]^{\mathcal{P}'}$ by Lemma 2, and so $\hat{t} \in \mathcal{O}^{\mathcal{P}'}(g_t\ e) = \mathcal{O}^{\mathcal{P}'}(g_t\ e')$ by definition, Th. 2 and hypothesis. But then $\hat{t} \in [\![g_t\ e']\!]^{\mathcal{P}'}$ by definition and Th. 2, and so $t \in [\![e']\!]^{\mathcal{P}}$ by Lemma 2 again. The other inclusion of $[\![e']\!]$ in $[\![e]\!]$ can be proved in a similar way.

Now we will prove the full abstraction wrt. $\mathcal{O}_{fo}$. The left to right implication can be proved in exactly the same way we did for $\mathcal{O}$. Concerning the other implication we modify the transformation $\hat{\_}$ of Lemma 2 in the following way:

$$
\begin{array}{l}
h\ \widehat{t_1\ \ldots} t_m = h_m\ \hat{t_1}\ \ldots\ \hat{t_m} \\
g_{(h\ t_1\ \ldots t_m)}(h\ X_1\ \ldots X_m) \to h_m\ (g_{t_1} X_1) \ldots (g_{t_m} X_m)
\end{array}
$$

where h_m is a fresh constructor symbol of arity m. Note that then $\forall t \in Pat_\perp$ we have $\hat{t} \in FOPat$. Besides it is still easy to prove that for any $e \in Exp_\perp, t \in Pat_\perp$ built with the signature of $\mathcal{P}$, $t \in [\![e]\!]^{\mathcal{P}}$ iff $\hat{t} \in [\![g_t\ e]\!]^{\mathcal{P}'}$, where $\mathcal{P}' = \mathcal{P} \uplus \mathcal{P}_{g_t}$, and that $\mathcal{P}'$ is a safe extension of $\mathcal{P}$, by a trivial modification of the proof for Lemma 2. With these tool the proof proceeds exactly like in the one for $\mathcal{O}$, but using these new definitions of $\hat{\_}$ and g_t.

3.2 Discussion: the case of extra variables

As pointed in Sect. 2, in this work we assume that our programs do not contain extra variables, i.e., $var(r) \subseteq var(f\ \overline{t})$ holds for any program rule $f\ t_1 \ldots t_n \to r$.

This condition is necessary for the full abstraction results to hold, as we can see in the following example.

Example 2. Consider a signature such that $FS = \{f/1, g/1\}$, $CS = \{0/0, 1/0\}$, and the program $\mathcal{P} = \{f\ X \to Y\ X\}$. Note the extra variable Y in the rule for f.

Then we have $[\![f\ 0]\!]^{\mathcal{P}} = \{\bot\} = [\![f\ 1]\!]^{\mathcal{P}}$, because any derivation of $f\ 0 \twoheadrightarrow t$ using (OR) must have the form

$$\dfrac{0 \twoheadrightarrow 0 \quad \dfrac{\cdots}{\varphi\ 0 \twoheadrightarrow t}\ X}{\mathcal{P} \vdash f\ 0 \twoheadrightarrow t}\ OR$$

where φ can be any pattern (f, g, 0, 1 or $\bot$) and X can be (OR) or (B). In all cases the only possible value for t in $\varphi\ 0 \twoheadrightarrow t$ will be $\bot$. A similar reasoning holds for $f\ 1$. However, for $\mathcal{P}' = \mathcal{P} \uplus \{g\ 0 \to 1\}$, which is a safe extension for $(\mathcal{P}, f\ 0)$ and $(\mathcal{P}, f\ 1)$ we can do:

$$\dfrac{0 \twoheadrightarrow 0 \quad \dfrac{0 \twoheadrightarrow 0 \quad 1 \twoheadrightarrow 1}{g\ 0 \twoheadrightarrow 1}\ OR}{\mathcal{P}' \vdash f\ 0 \twoheadrightarrow 1}\ OR$$

while for $f\ 1$ we can only do:

$$\dfrac{1 \twoheadrightarrow 1 \quad \dfrac{}{g\ 1 \twoheadrightarrow \bot}\ B}{\mathcal{P}' \vdash f\ 1 \twoheadrightarrow \bot}\ OR$$

Hence the context $[\,]$ and the safe extension $\mathcal{P}'$ yield different observations for $f\ 0$ and $f\ 1$.

The previous example can be discarded if we assume that we have at least one constructor for each arity, or at least for the maximum of the arities of function symbols. This is reasonable because it is like having tuples of any arity. With this assumption and the previous program and expression we do not have $[\![f\ a]\!]^{\mathcal{P}} = [\![f\ b]\!]^{\mathcal{P}}$ anymore, as $c\ a \in [\![f\ a]\!]$ and $c\ b \in [\![f\ b]\!]$, hence the hypothesis of the condition for full abstraction fails.
Nevertheless the following example shows that full abstraction fails even under the assumption of having a constructor for each arity.

Example 3. For $\mathcal{P} = \{f\ 1 \to 2, h\ X \to f\ (Y\ X)\}$ and $FS = \{f/1, h/1, g/1\}$ we have $\forall \theta \in PSubst_{\bot}, 1 \notin [\![(\theta(Y))\ 0]\!]^{\mathcal{P}} \cup [\![(\theta(Y))\ 1]\!]^{\mathcal{P}}$, hence $[\![h\ 0]\!]^{\mathcal{P}} = \{\bot\} = [\![h\ 1]\!]^{\mathcal{P}}$. But for $\mathcal{P}' = \mathcal{P} \uplus \{g\ 0 \to 1\}$, which is a safe extension for $(\mathcal{P}, h\ 0)$ and $(\mathcal{P}, h\ 1)$, we have $\mathcal{P}' \vdash h\ 0 \twoheadrightarrow 2$ while $\mathcal{P}' \vdash h\ 1 \not\twoheadrightarrow 2$.

The point is that, if extra variables are allowed, for a fixed program $\mathcal{P}$ and an expression e we cannot ensure that for any safe extension $\mathcal{P}'$ for $(\mathcal{P}, e)$ it holds that $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$; i.e., Lemma 1 does not hold. We cannot even grant that $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ implies that $[\![e]\!]^{\mathcal{P}'} = [\![e']\!]^{\mathcal{P}'}$ for any safe extension $\mathcal{P}'$, which in fact is what it is needed for full abstraction, and what we have exploited in

the (counter-)examples above. It is also relevant that both examples are left-FO programs, and therefore the problems do not come from the presence of higher order patterns in function definitions.

As a conclusion of this discussion, we contemplate the extension of this work to cope with extra variables as a challenging subject of future work.

4 Conclusions and Future Work

We have seen that reasoning extensionally in existing FLP languages with HO nondeterministic functions is not valid in general (Ex. 1, Prop. 2). In contrast, thinking in intensional functions is not an arbitrary exoticism, but rather an appropriate point of view for that setting (Th. 3). We stress the fact that adopting an intensional view of the *meaning* of functions is compatible with a discipline of programming in which programs are restricted to be left-FO, that is, the use of HO-patterns in left-hand sides of program rules is forbidden. This is the preferred choice by some people in the FLP community, mostly because HO-patterns in left-hand sides cause some problems to the type system. Our personal opinion is the following: since HO-patterns appear in the semantics even if they are precluded from programs, they could be freely permitted, at least as far as they are compatible with the type discipline. There are quite precise works [8] pointing out which are the problematic aspects, mainly *opacity* of patterns. Existing systems could incorporate restrictions, so that only type-safe uses of HO-patterns are allowed. More work could be done along this line.

We have seen in Sect. 3.2 how the presence of extra variables in programs destroys full-abstraction of the *HOCRWL* semantics. Recovering it for such family of programs is an obvious subject of future work. Another very interesting, and somehow related matter, is giving variables a more active role in the semantics. Certainly, the results in the paper are not restricted to ground expressions, but their interest for expressions having variables is limited by the fact that in the notions of semantics and observations considered in the paper, variables are implicitly treated as generic constants. For instance, the expressions $e_1 \equiv X + X$ and $e_2 \equiv X + 0$ do have the same semantics $[\![\_]\!]_\perp$ ($[\![e_1]\!]_\perp = [\![e_2]\!]_\perp = \{\perp\}$). Full abstraction of $[\![\_]\!]_\perp$ ensure that $\mathcal{O}(\mathcal{C}[e_1]) = \mathcal{O}(\mathcal{C}[e_2])$ for any context $\mathcal{C}$. This is ok as far as one is only interested in possible reductions starting from e_1, e_2. If this is the case, certainly e_1 and e_2 have equivalent behavior (no successful reduction to a pattern can be done with any of them). However, in some sense e_1 and e_2 have different 'meanings', that are reflected in different behaviors; for instance, if e_1 and e_2 are subject to narrowing, or if e_1 and e_2 are used as right hand sides in a program rule.

Acknowledgments We are grateful to Rafa Caballero for his intense collaboration while developing this research.

12

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs . In *ICLP*, pages 87–101, 2006.
3. G. Berry, P. Curien, and J. Levy. Full abstraction for sequential languages: the state of the art. In *Algebraic methods in semantics*, pages 89–132. Cambridge University Press, New York, NY, USA, 1986.
4. J. Dios and F. López-Fraguas. Elimination of extra variables from functional logic programs. In P. Lucio and F. Orejas, editors, *VI Jornadas sobre Programación y Lenguajes (PROLE 2006)*, pages 121–135. CINME, 2006.
5. R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.
6. R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.
7. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.
8. J. González-Moreno, T. Hortalá-González, and Rodríguez-Artalejo, M. Polymorphic types in functional logic programming. In *Journal of Functional and Logic Programming*, volume 2001/S01, pages 1–71, 2001. Special issue of selected papers contributed to the International Symposium on Functional and Logic Programming (FLOPS'99).
9. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
10. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
11. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
12. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
13. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
14. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Equivalence of two formal semantics for functional logic programs. *Electronic Notes in Theoretical Computer Science 188*, pages 117–142, 2007.
15. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
16. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
17. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

18. J. C. Mitchell. *Foundations of programming languages*. MIT Press, Cambridge, MA, USA, 1996.
19. G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):225–255, 1977.

7.2.6 A Formalization of the Semantics of Functional-Logic Programming in Isabelle

A Formalization of the Semantics of Functional-Logic Programming in Isabelle*

Francisco J. López-Fraguas[1], Stephan Merz[2], and Juan Rodríguez-Hortalá[1]

[1] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, juanrh@fdi.ucm.es
[2] INRIA Nancy & LORIA
Stephan.Merz@loria.fr

Abstract. Modern functional-logic programming languages like Toy or Curry feature non-strict non-deterministic functions that behave under call-time choice semantics. A standard formulation for this semantics is the *CRWL* logic, that specifies a proof calculus for computing the set of possible results for each expression. In this paper we present a formalization of that calculus in the Isabelle/HOL proof assistant. We have proved some basic properties of *CRWL*: closedness under c-substitutions, polarity and compositionality. We also discuss some insights that have been gained, such as the fact that left linearity of program rules is not needed for any of these results to hold.

1 Introduction

Fully formalizing the (meta)theory of a programming language can be beneficial for developing its foundations. There is an increasing number of researchers (see e.g. [2]) sharing the conviction that the combination *formalization+mechanized theorem proving* must (and will) play a prominent role in programming languages research and technology. In particular, formalizations help to clarify overlooked aspects, to discover pitfalls, and even to provide new insights; moreover, formalized metatheories lead to mechanized reasoning about programs, giving reliable support to tools like certifying compilers or certified program transformations.

In this paper we formalize the semantics of functional logic programming (FLP), a well established paradigm (see [9]) integrating features of logic and functional languages. In modern FLP languages such as Curry [10] or Toy [14] programs are constructor based rewrite systems that may be non-terminating and non-confluent. Semantically this leads to the presence of non-strict and non-deterministic functions. The semantics adopted for non-determinism is *call-time choice* [11, 8], informally meaning that in any reduction, all descendants of a given subexpression must share the same value. The semantic framework *CRWL*[3] was proposed in [7, 8] to accomodate this view of non-determinism, and

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM), S-0505/TIC/0407 (PROMESAS-CAM) and TIN2008-06622-C03-01/TIN (FAST-STAMP).

[3] *CRWL* stands for "Constructor-based ReWriting Logic".

is nowadays considered the standard semantics of FLP. For the purpose of this paper, the most relevant aspect of *CRWL* is a proof calculus devised to prove reduction statements of the form $\mathcal{P} \vdash e \twoheadrightarrow t$, meaning that t is a possible (partial) value to which e can be reduced using the program $\mathcal{P}$.

We have chosen Isabelle/HOL as concrete logical framework for our formalization. Using such a broadly used system is not only easier, but also more flexible and stable than developing language specific tools like has been done, e.g., for logic programming [15] or functional programming [6].

The remainder of the paper is organized as follows: Sect. 2 contains some preliminaries about the *CRWL* framework , Sect. 3 presents the Isabelle theories developed to formalize *CRWL*, and Sect. 4 gives the mechanized proofs of some important properties of *CRWL*. Finally, Sect. 5 summarizes some conclusions and points to future work.

An extended version of this paper can be found at `http://gpd.sip.ucm.es/juanrh/pubs/isabell-crwl-report.pdf`. The Isabelle code underlying the results presented here is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/IsabelleCrwl`.

2 Preliminaries

2.1 Constructor-based term rewrite systems

We consider a first-order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *constructor* and defined *function* symbols respectively, each with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. The set *Exp* of *expressions* is inductively defined as

$$Exp \ni e ::= X \mid h(e_1, \ldots, e_n),$$

where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set *CTerm* of *constructed terms* (or *c-terms*) is defined like *Exp*, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that *Exp* stands for evaluable expressions, i.e., expressions that can contain function symbols, while *CTerm* stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as

$$Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$$

for $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written $\mathcal{C}[e]$, is defined inductively by

$$[\,][e] = e \quad \text{and} \quad h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n).$$

The set *Subst* of *substitutions* consists of finite mappings $\theta : \mathcal{V} \longrightarrow Exp$ (i.e., mappings such that $\theta(X) \neq X$ only for finitely many $X \in \mathcal{V}$), which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e, and $\theta\theta'$

$$\textbf{(RR)}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad\qquad \textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot}$$

$$\textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\quad c \in CS^n$$

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow p_1\theta \ldots e_n \twoheadrightarrow p_n\theta \quad r\theta \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t}\quad \begin{array}{l} f(p_1,\ldots,p_n) \to r \in \mathcal{P} \\ \theta \in CSubst_\bot \end{array}$$

Fig. 1. Rules of *CRWL*

for the composition of substitutions, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain of θ is defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$. In most cases we will use *c-substitutions* $\theta \in CSubst$, for which $X\theta \in CTerm$ for all $X \in dom(\theta)$.

A CRWL-*program* (or simply a *program*) is a set of rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that each variable occurs only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. *CRWL*-programs often allow also conditions in the program rules. However, *CRWL*-programs with conditions can be transformed into equivalent programs without conditions, therefore we consider only unconditional rules.

2.2 The *CRWL* framework

In order to accomodate non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot$, $CTerm_\bot$, $Subst_\bot$, $CSubst_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying

$$\bot \sqsubseteq e \quad \text{and} \quad e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e'] \text{ for all } e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$$

This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus (see Fig. 1) for deriving reduction statements $\mathcal{P} \vdash e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, meaning informally that t is (or approximates) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice. Rule B (bottom) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) allow us to reduce any variable to itself, and to decompose the evaluation of an expression whose root symbol is a constructor. Rule OR (outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of a $CSubst_\bot$ θ) and then reduce the instantiated right-hand side. The use of partial c-substitutions in OR is essential to express call-time choice, as only single partial values are used for parameter passing. Notice also that by the effect of θ in OR

extra variables in the right-hand side of a rule can be replaced by any c-term, but not by any expression. The *CRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$.

3 Formalizing *CRWL* in Isabelle

3.1 Basic definitions

We describe our formalization of *CRWL* in Isabelle. The first step is to define elementary types for the syntactic elements.

```
datatype signat = fs string | cs string
datatype varId = vi string
datatype exp = perp | Var varId | Ap signat "exp list"
types
  subst = "varId ⇒ exp option"
  rule = "exp * exp"
  program = "rule set"
```

Signatures are represented by a datatype that provides two constructors `cs` and `fs` to distinguish between constructor and function symbols. The type `varId` is used to represent variable identifiers, which will be employed to define substitutions. Then the datatype `exp` is naturally defined following the inductive scheme of $Exp_\bot$, therefore with this representation every expression is partial by default.

Substitutions (type `subst`) are represented as partial functions from variable identifiers to expressions, using Isabelle's `option` type. Hence the domain of a substitution ϑ will be the set of elements from `varId` for which ϑ returns some value different from `None`. Note that this representation does not ensure that domains of substitutions are finite. Our proofs do not rely on this finiteness assumption. Finally we represent a program rule as a pair of expressions, where the first element is considered the left-hand side of the rule and the second the right-hand side, and a program simply as a set of program rules. The set of valid *CRWL* programs is characterized by a predicate `crwlProgram :: "program ⇒ bool"` that checks whether the restrictions of left-linearity and constructor discipline are satisfied.

We define a function `apSubst :: "subst ⇒ exp ⇒ exp"` for applying a substitution to an expression. The composition of substitutions is defined through a function `substComp :: "subst ⇒ subst ⇒ subst"`. The following lemma ensures the correctness of this definition.

```
lemma subsCompAp :
  "(apSubst ϑ (apSubst σ e)) = (apSubst (substComp ϑ σ) e)"
```

Just as ML, the Isabelle type system does not support subtyping, which could have been useful to represent the sets of c-terms and c-substitutions. Instead, we define predicates `cterm` and `csubst` characterizing these subtypes. We prove the expected lemmas, such as that the composition of two c-substitutions is a c-substitution, or that the application of a c-substitution to a c-term yields a c-term.

3.2 Approximation order and contexts

Two key notions of *CRWL* have not yet been formalized: the approximation order $\sqsubseteq$, which will be used in the formulation of the polarity of *CRWL*, and the notion of one-hole context, which will be used in the compositionality.

The following inductively defined predicate `ordap` (with concrete infix syntax $\sqsubseteq$) models the approximation order.

```
inductive
  ordap :: "exp ⇒ exp ⇒ bool" ("_ ⊑ _" [51,51] 50)
where
  B: "perp ⊑ e"
| V: "Var x ⊑ Var x"
| Ap: "⟦ size es = size es' ; ALL i < size es. es!i ⊑ es'!i ⟧
       ⟹ Ap h es ⊑ Ap h es'"
```

Rule `B` asserts that `perp` $\sqsubseteq$ `e` holds for every `e`; rule `V` is needed for $\sqsubseteq$ to be reflexive; finally rule `Ap` ensures closedness under Σ-operations, and thus compatibility with context [3], because $\sqsubseteq$ is reflexive and transitive, as we will see. The following results state that our formulation of $\sqsubseteq$ defines a partial order.

```
lemma ordapRefl : "e ⊑ e"
lemma ordapTrans :
  assumes "e1 ⊑  e2" and "e2 ⊑ e3"
  shows "e1 ⊑  e3"
lemma ordapAntisym :
  assumes "e1 ⊑ e2" and "e2 ⊑ e1"
  shows "e1 = e2"
definition ordap_less ("_ ⊏ _" [51,51] 50) where
  "e ⊏ e' ≡ e ⊑ e' ∧ e ≠ e'"
interpretation exp : order [ordap ordap_less]
```

Contexts are represented as the datatype `cntxt`, defined as follows:

```
datatype cntxt = Hole | Cperp | CVar varId
               | CAp signat "cntxt list"
```

Note that `cntxt` cannot follow the inductive structure of *Cntxt* with precision, because the type system of Isabelle is not expressive enough to allow us to specify that only one of the arguments of `CAp` will be a context and the others will be expressions. Then our contexts are defined as expressions with possibly some holes inside. Therefore the datatype `cntxt` represents contexts with any number of holes, even zero holes, and the function `apCon :: "exp ⇒ cntxt ⇒ exp"` is defined so it puts the argument expression in every hole of the argument context. In order to characterize contexts with just one hole, we define a function `numHoles :: "cntxt ⇒ nat"` that returns the numbers of holes in a context. Using it we can define define predicates `oneHole` and `noHole` and prove the following lemmas.

```
lemma noHoleApDontCare :
  assumes "noHole xC"
  shows "apCon e xC = apCon e' xC"

lemma oneHole :
  assumes "oneHole (CAp h xCs)"
  shows "∃ xC yCs zCs. xCs = (yCs @ xC # zCs) ∧ oneHole xC ∧
                       (∀c ∈ set (yCs @ zCs). noHole c)"
```

3.3 The *CRWL* logic in Isabelle/HOL

The *CRWL* logic has been formalized through the inductive predicate `clto` with infix notation `"_ ⊢ _ → _"`. The rules defining `clto` faithfully follow the inductive structure of the definition of *CRWL* as it is presented in Fig. 1.

```
inductive
  clto :: "program ⇒ exp ⇒ exp ⇒ bool"  ("_ ⊢ _ → _"
[100,50,50] 38)
where
  B[intro]:  "prog ⊢ exp → perp"
| RR[intro]: "prog ⊢ Var v → Var v"
| DC[intro]: "⟦size es = size ts;
               ∀i < size es. prog ⊢ es!i → ts!i
              ⟧ ⟹ prog ⊢ Ap (cs c) es → Ap (cs c) ts"
| OR[intro]: "⟦(Ap (fs f) ps, r) ∈ prog ; csubst ϑ ;
               size es = size ps ;
               ∀i < size es. prog ⊢ es!i → apSubst ϑ (ps!i);
               prog ⊢ apSubst ϑ r → t
              ⟧ ⟹ prog ⊢ Ap (fs f) es → t"
```

Using `clto` we can easily define the *CRWL* denotations in Isabelle as follows.

```
definition den :: "program ⇒ exp ⇒ exp set" where
  "den P e = {t. P ⊢ e → t}"
```

4 Reasoning about *CRWL* in Isabelle

The first interesting property that we are proving about *CRWL* expresses that evaluation is *closed under c-substitutions*: reductions are preserved when terms are instantiated by c-substitutions.

```
theorem crwlClosedCSubst :
  assumes "prog ⊢ e → t" and "csubst ϑ"
  shows "prog ⊢ apSubst ϑ e → apSubst ϑ t"
```

The proof of this lemma proceeds by induction on the *CRWL*-proof of the hypothesis, therefore we will have one case for each *CRWL* rule. The first three cases are proved automatically. However, to prove the case for rule `OR` Isabelle needs some help from us. We need to prove

```
prog ⊢ (Ap (fs f) (map (apSubst ϑ) es)) → (apSubst ϑ t)
```

and then let the simplifier apply the definition of `apSubst`. In the proof for that subgoal we used lemma `CSubsComp` to ensure that the c-substitution μ used for parameter passing composed with the c-substitution ϑ in the hypothesis yields another c-substitution, and lemma `subsCompAp` to guarantee the correct behaviour of the composition for those c-substitutions.

Note that for this result to hold no additional hypotheses about the program or the expressions involved are needed. In particular, this implies that the result holds even for programs that do not follow the constructor discipline or that have non left-linear rules. The Isabelle proof clearly shows that the important ingredients are the use of c-substitutions for parameter passing and the reflexivity of *CRWL* wrt. c-terms, expressed by lemma `ctermRefl`, which allows us to reduce to itself any expression `X`ϑ coming from a premise `X` $\rightarrow$ `X`.

The second property that we address is the *polarity of* CRWL. This property is formulated by means of the approximation order and roughly says that if we can compute a value for an expression then we can compute a smaller value for a bigger expression. Here we should understand the approximation order as an information order, in the sense that the bigger the expression, the more information it gives, and so more values can be computed from it.

```
theorem crwlPolarity :
  assumes "prog ⊢ e → t" and "e ⊑ e'" and "t' ⊑ t"
  shows "prog ⊢ e' → t'"
using assms proof (induct arbitrary: e' t')
```

The idea of the proof is to construct a *CRWL*-proof for the conclusion from the *CRWL*-proof of the hypothesis, hence it is natural to proceed by induction on the structure of this proof (method `induct`). The qualifier `arbitrary` is used to generalize the assertion for any expressions `e'` and `t'`. The proof also relies on the following additional lemmas about the approximation order, which were proved automatically by Isabelle.

```
lemma ordapPerp: assumes "e ⊑ perp" shows "e = perp"
lemma ordapVar: assumes "Var v ⊑ e" shows "e = Var v"
lemma ordapVar_converse:
   assumes "e ⊑ Var v" shows "e = perp ∨ e = Var v"
lemma ordapAp:
   assumes "Ap h es ⊑ e'"
   shows "∃es'. e' = Ap h es' ∧ size es = size es'
                ∧ (ALL i < size es. es!i ⊑ es'!i)"
lemma ordapAp_converse:
   assumes "e' ⊑ Ap h es"
   shows "e' = perp ∨
          (∃es'. e' = Ap h es' ∧ size es = size es'
                  ∧ (ALL i < size es. es'!i ⊑ es!i))"
```

The inductive proof for theorem `crwlPolarity` again considers each *CRWL* rule in turn. In the case for `B` we have `t = perp`, hence we just have to apply `ordapPerp` to get `t' = perp`, and then use the *CRWL* rule `B`. Regarding `RR`, as

then `t = Var v`, by `ordapVar_converse` we get that either `t' = perp` or `t' = Var v`. The first case is trivial, and in the latter we just have to apply `ordapVar` getting `e' = Var v`, which is enough for Isabelle to finish the proof automatically. The case of `DC` is more complicated. Again we obtain two cases for `t' = perp` and `t'` a constructor application, by using lemma `ordapAp_converse`. While the first case is trivial, the second one requires some involved reasoning over the list of arguments, using the information we get from applying lemma `ordapAp`. Finally, the proof for `OR` is similar to the second case of the proof for `DC`, with a similar manipulation of the list of arguments, and the use of lemma `ordapAp` to obtain the induction hypothesis for the arguments.

Once again we find that this proof does not require any hypothesis on the linearity or the constructor discipline of the program: this is indeed quite obvious because this property only talks about what happens when we replace some subexpression by `perp`.

Finally we will tackle the *compositionality of* CRWL, that says that if we take a context with just one hole and an expression, then the set of values for the expression put it that context will be the union of the set of values for the result of putting each value for the expression in that context.

```
theorem compCRWL :
  assumes "oneHole xC"
  shows "den P (apCon e xC) =
         (⋃t∈den P e. den P (apCon t xC))"
```

We have proved the two set inclusions separately as auxiliary lemmas `compCRWL1` and `compCRWL2`. The proofs of these lemmas are quite laborious but essentially proceed by induction on the *CRWL*-proof in their hypothesis, using it to build a *CRWL*-proof for the statement in the conclusion. In these proofs, Lemma `noHoleApDontCare` from Subsect. 3.2 is fundamental.

Again, while theorem `compCRWL` requires the context to have just one hole, it does not assume the linearity or constructor discipline of the program. This came as a surprise to us, and initially made us doubt about the accuracy of our formalization of *CRWL*. But it turns out that although *CRWL* is designed to work with *CRWL*-programs, that fulfil these restrictions, it can also be applied to general programs. For those programs some properties, such as the theorems `crwlClosedCSubst`, `crwlPolarity`, and `compCRWL` still hold, but other fundamental properties do not, in particular the strong adequacy results w.r.t. its operational counterparts of [8, 12, 1]. The point is that for those programs *CRWL* does not deliver the "intended semantics" anymore. And this is not strange, because that semantics was intended with *CRWL*-programs in mind. For example, consider the non linear program $\mathcal{P} = \{f(X, X) \to a\}$. There is a *CRWL*-proof for the statement $\mathcal{P} \vdash f(a, b) \twoheadrightarrow a$ but this value cannot be computed in any of the operational notions of [8, 12, 1] nor in any implementation of FLP, in which the independence of the matching process of the arguments — derived from left-linearity of program rules — is assumed. It is also not very natural that $f(a, b)$ could yield the value a for the arguments a and b being different values, which implies that the semantics defined by *CRWL* for non left-linear

programs is pretty odd. But that is not a big problem, because we only care about the properties of *CRWL* for the kind of programs it has been designed to work with. And if it enjoys some interesting properties for a bigger class of programs that is fine, because that nice properties will be inherited by the class of *CRWL*-programs.

On the other hand, for programs not following the constructor discipline, we will not even be able to have a matching for an argument of a rule which is not a constructor, because in the rule OR we have to reduce every argument of a function call to a value, which will be a c-term by Lemma `ctermVals` (see the extended version of this paper), and so could never be an instance of expression containing function symbols. Thus, the rule OR could not be used for program rules not following the constructor discipline.

5 Conclusions

This paper presented a formalization of the essentials of *CRWL* [7, 8], a well-known semantic framework for functional logic programming, in the interactive proof assistant Isabelle/HOL. We chose that particular logical framework for its stability and its extensive libraries. The Isar proof language allowed us to structure the proofs so that they become quite elegant and readable, as can be observed by looking at the Isabelle code.

Our formalization is generic with respect to syntax, and includes important auxiliary notions like substitutions or contexts. This is in contrast to previous work [4, 5] that focused on formalizing the semantics of each concrete program. In contrast, our paper focuses on developing the metatheory of the formalism, allowing us to obtain results that are more general and also more powerful: we formally prove essential properties of the paradigm like *polarity* or *compositionality* of the *CRWL*-semantics. We plan to extend our theories so that we will be able to reason about properties of concrete programs by deriving theorems that express verification conditions in the line of those stated in [4, 5].

While developing the formalization we realized an interesting fact not pointed out before: properties like polarity or compositionality do not depend on the constructor discipline and left-linearity imposed to programs. However, such requirements will certainly play an essential role when extending our work to formally relate the *CRWL*-semantics with operational semantics like the one developed in [12], one of our intended subjects of future work. We think that could be interesting in several ways. First of all it would be a further step in the direction of challenge 3 of [2], "Testing and Animating wrt the Semantics", because we would end up getting an interpreter of *CRWL* during the process. We should then also formalize the evaluation strategy for the operational semantics, obtaining an Isabelle proof of its optimality. Finally there are precedents [13, 12] of how the combination of a denotational and operational perspective is useful for general semantic reasoning in FLP.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998.
4. J. Cleva, J. Leach, and F. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'04*, pages 9–19. ACM, 2004.
5. J. Cleva and I. Pita. Verification of CRWL programs with rewriting logic. *J. Universal Computer Science*, 12(11):1594–1617, 2006.
6. M. de Mol, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Theorem proving for functional programmers. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2002*, volume 2312 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2001.
7. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.
8. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
9. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
10. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
11. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
12. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming*, pages 197–208. ACM Press, 2007.
13. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
14. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
15. R. F. Stärk. The theoretical foundations of lptp (a logic program theorem prover). *J. Log. Program.*, 36(3):241–269, 1998.

Part III

Extended Versions

Chapter 8

Extended versions

8.1.7 A Simple Rewrite Notion for Call-time Choice Semantics (Extended version)

A Simple Rewrite Notion for Call-time Choice Semantics *

(Extended version: revision March 2010)

Francisco J. López-Fraguas Juan Rodríguez-Hortalá Jaime Sánchez-Hernández
Dep. Sistemas Informáticos y Computación, Universidad Complutense de Madrid
fraguas@sip.ucm.es jrodrigu@fdi.ucm.es jaime@sip.ucm.es

Abstract

Non-confluent and non-terminating rewrite systems are interesting from the point of view of programming. In particular, existing functional logic languages use such kind of rewrite systems to define possibly non-strict non-deterministic functions. The semantics adopted for non-determinism is call-time choice, whose combination with non-strictness is not a trivial issue that has been addressed from a semantic point of view in the Constructor-based Rewriting Logic (*CRWL*) framework. We investigate here how to express call-time choice and non-strict semantics from a point of view closer to classical rewriting. The proposed notion of rewriting uses an explicit representation for sharing with *let*-constructions and is proved to be equivalent to the *CRWL* approach. Moreover, we relate this *let*-rewriting relation (and hence *CRWL*) with ordinary rewriting, providing in particular soundness and completeness of *let*-rewriting with respect to rewriting for a class of programs which are confluent in a certain semantic sense.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Theory, languages.

Keywords Functional-logic programming, term rewriting systems, constructor-based rewriting logic, non-determinism, call-time choice semantics, sharing, local bindings.

1. Introduction

Modern functional logic programs as considered in systems like *Curry* [12] or *Toy* [17] are constructor-based term rewrite systems, possibly non-terminating and non-confluent, thus defining possibly non-strict non-deterministic functions, as happens with the program in Figure 1.

The semantics adopted for non-determinism in those systems is *call-time choice* semantics [10, 13], also called sometimes *singular* semantics [25]. Loosely speaking, call-time choice conceptually means to pick a value for each argument of a function application before applying it. Call-time choice is easier to understand and implement in combination with strict semantics and eager evaluation in terminating systems as in [13], but can be made also compatible –via partial values and sharing– with non-strictness and laziness in the presence of non-termination.

In the example of Figure 1 the expression $heads(repeat(coin))$ can take, under call-time choice, the values $(0, 0)$ and $(1, 1)$, but not $(0, 1)$ or $(1, 0)$. The example illustrates also a key point here, that ordinary term rewriting is an unsound procedure for call-time choice semantics with non-determinism, since a possible rewrite is

$$heads(repeat(coin)) \rightarrow heads(coin : repeat(coin)) \rightarrow heads(0 : repeat(coin)) \rightarrow heads(0 : coin : repeat(coin)) \rightarrow heads(0 : 1 : repeat(coin)) \rightarrow (0, 1)$$

In operational terms, call-time choice would have required to share the value for all the occurrences of $coin$ in the reduction above.

It is commonly accepted (see e.g. [11]) that call-time choice semantics combined with non-strict semantics is adequately formally expressed by the *CRWL* framework [9, 10]. An additional indication of the usefulness of *CRWL* is the large set of its extensions that have been devised to cope with relevant aspects of declarative programming: higher order functions, types, constraints, constructive failure, ...(see [22] for a survey of the first works on the *CRWL* approach). However, a drawback of the *CRWL*-logic is its lack of a proper one-step reduction mechanism close both to the logic and to the computations, that could play a role similar to rewriting with respect to equational logic. Certainly *CRWL* includes operational procedures in the form of lazy narrowing based goal-solving calculi [10, 26], but they are too complex to be seen as the basic or 'fundamental' way to explain or understand how reduction can proceed in the presence of non-strict non-deterministic functions with call-time choice semantics.

Therefore, other works have been more influential on the operational side of the field, specially those based on the notion of needed narrowing [4], whose underlying theory is classical rewriting. Needed narrowing has become the 'official' operational procedure of functional logic languages, and has also been subject of various variations and improvements (see [11]).

These two coexisting branches of research (one based on *CRWL*, and the other based on classical rewriting via needed narrowing) have remained disconnected from the technical point of view, despite the fact that they both refer to what intuitively is the same programming language paradigm, as believed by most –if not all– people in the field.

This is not a satisfactory situation, because it precludes the possibility of applying –on a sound technical basis– results, notions and techniques from the semantic side to the operational side and viceversa. Our aim in this work is to establish that missing bridge.

A major problem is that needed narrowing adopts classical rewriting as underlying theory and therefore is not valid for call-time choice with non-determinism. This is overcome in practice by adding a sharing mechanism to the encoding of narrowing, but this is an implementation patch that is not enough for our technical purposes. Is there an existing notion of rewriting that can be used instead? Of course, the issue of combining sharing with rewriting or other reductions notions is not new. But a review of the literature (in Section 7 we make a short summary) suggested to us that there was still room for proposing a new formulation of rewriting tailored

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM) and S-0505/TIC/0407 (PROMESAS-CAM).

$coin \to 0$	$repeat(X) \ \to \ X{:}repeat(X)$
$coin \to 1$	$heads(X{:}Y{:}Ys) \ \to \ (X,Y)$

Figure 1. A non-terminating and non-confluent program

to call-time choice as realized by functional logic languages, and trying to fulfil the following requirements:
• it should be based on a notion of rewrite step, as to be useful to follow how a computation proceeds step by step.
• it should be simple enough to be easily understandable for non-expert potential users. (e.g., students) of functional logic languages adopting call-time choice.
• it should be provably equivalent to *CRWL*.
• it should serve as a basis of subsequent notions of narrowing and narrowing strategies.

We propose then a simple variant of rewriting that uses local bindings in the form of *let*-expressions to express sharing. Not surprisingly, our *let*-rewriting is very close to existing formalisms to express sharing in different contexts, like in [19] for λ-calculus, or term graph rewriting [21]. We are also inspired by [18] where indexed unions of set expressions – a construction generalizing the idea of *let*-expressions – were used to express sharing in an extension of *CRWL* to deal with constructive failure.

We also investigate the connection between our *let*-rewriting relation and classical rewriting. As we will prove, in general *let*-rewriting is sound with respect to rewriting, and is also complete for confluent systems (more precisely, for deterministic programs, a semantic property close to confluence).

The rest of the paper is organized as follows. Section 2 presents some preliminaries about term rewriting and the *CRWL* framework. Section 3 contains a first discussion about how to express non-strict call-time choice by rewriting. Section 4 introduces local bindings in syntax to express sharing and defines *let*-rewriting as an adequate notion of rewriting for them. In Section 5 we prove the equivalence of *CRWL* and *let*-rewriting. In Section 6 we address the relationship between of *let*-rewriting and classical rewriting, proving in particular their equivalence for deterministic programs. Finally, Section 7 reviews related work and summarizes some conclusions. Full proofs can be found at the appendix.

2. Preliminaries

2.1 Constructor based term rewriting systems

We assume a fixed first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of constructor and defined function symbols respectively, each of them with an associated arity; we write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. As usual notations we write $c, d \ldots$ for constructors, $f, g \ldots$ for functions and $x, y \ldots$ for variables taken from a numerable set $\mathcal{V}$.

To avoid confusion with the usual terminology of *CRWL* (introduced below) we follow its approximation introducing two kinds of syntactic objects: expressions and terms. The set Exp of *expressions* is defined as $Exp \ni e ::= x \mid h(e_1, \ldots, e_n)$, where $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) has the same definition of Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain (user-defined) function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, t', s' \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$.

Contexts (with one hole) are defined by $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, where $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written as $\mathcal{C}[e]$, is defined inductively by $[\,][e] = e$; $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

Substitutions are mappings $\theta : \mathcal{V} \longrightarrow Exp$ which extend naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e. The domain and range of θ are defined as $dom(\theta) = \{x \in \mathcal{V} \mid x\theta \neq x\}$ and $ran(\theta) = \bigcup_{x \in dom(\theta)} var(x\theta)$. Given a set of variables D the notation $\theta|_D$ represents the substitution θ restricted to D and $\theta|_{\setminus D}$ is a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. A *c-substitution* is a substitution θ such that $x\theta \in CTerm$ for all $x \in dom(\theta)$. We write $Subst$ and $CSubst$ for the sets of substitutions and c-substitutions. Throughout the paper, the notation $\overline{o}$ stands for tuples of any of the previous syntactic construction o.

A *constructor based rewrite rule* (or c-rewrite rule) has the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear tuple of c-terms, where linear means that no variable occurs twice in the tuple. Notice that we allow e to have extra variables (i.e., variables not occurring in the left-hand side). A constructor-based rewrite system (or *c-rewrite system*) is a set of c-rewrite rules. Given a c-rewrite system $\mathcal{P}$, its rewrite relation $\to_{\mathcal{P}}$ is defined by $\mathcal{C}[l\theta] \to_{\mathcal{P}} \mathcal{C}[r\theta]$, for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and substitution θ. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. Since in this paper we only consider constructor based rules, we will often speak simply of rewrite rules or rewrite systems. Furthermore, we will usually omit the reference to $\mathcal{P}$ in $\to_{\mathcal{P}}$.

Confluence for constructor-based term rewrite systems is defined in the usual way: a program $\mathcal{P}$ is confluent if for any $e, e_1, e_2 \in Exp$ such that $e \overset{*}{\to}_{\mathcal{P}} e_1$, $e \overset{*}{\to}_{\mathcal{P}} e_2$ there exist $e_3 \in Exp$ such that both $e_1 \overset{*}{\to}_{\mathcal{P}} e_3$ and $e_2 \overset{*}{\to}_{\mathcal{P}} e_3$.

2.2 The CRWL framework

In the *CRWL* framework [9, 10], programs are c-rewrite systems, also called *CRWL-programs* (or simply 'programs') from now on. The original *CRWL* logic considered also the possible presence of *joinability* constraints as conditions in rules in order to give a better treatment of strict equality as built-in, which is a subject orthogonal to the aims of this paper. Furthermore, due to the semantic given to equality in functional logic and thanks to the allowance of extra variables in rules, it is possible to replace conditions by the use of an *if_then* function, as has been technically proved in [24] for *CRWL* and in [2] for term rewriting. Therefore, we consider only unconditional rules.

To deal with non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot, CTerm_\bot, Subst_\bot, CSubst_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, meaning informally that t is (or approximates to) a possible value of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice.

The *CRWL*-proof calculus is presented in Figure 2. Rule **(B)** allows any expression to be undefined or not evaluated (non-strict semantics). Rule **(OR)** expresses that to evaluate a function call we

must choose a compatible program rule, perform parameter passing (by means of a c-substitution θ) and then reduce the right-hand side. The use of c-substitutions in **(OR)** is essential to express call-time choice; notice also that by the effect of θ in **(OR)**, extra variables in the right-hand side of a rule can be replaced by any c-term, but not by any expression as in the notion of ordinary rewriting $\rightarrow_{\mathcal{P}}$.

We write $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in the *CRWL*-calculus using the program $\mathcal{P}$. Given a program $\mathcal{P}$, the *CRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}}_{CRWL} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$.

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(RR)}\ \frac{}{x \twoheadrightarrow x}\quad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1\ \ldots\ e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\quad c \in CS^n, t_i \in CTerm_\bot$$

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow t_1\theta\ \ldots\ e_n \twoheadrightarrow t_n\theta \quad e\theta \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t}\quad \begin{array}{l} f(\bar{t}) \rightarrow e \in \mathcal{P} \\ \theta \in CSubst_\bot \end{array}$$

Figure 2. Rules of *CRWL*

As an example, Figure 3 shows a *CRWL*-derivation for the statement $heads(repeat(coin)) \twoheadrightarrow (0,0)$, using the program of Figure 1. Observe that in the derivation there is only one reduction statement for $coin$ (namely $coin \twoheadrightarrow 0$), and the obtained value 0 is then *shared* in the whole derivation, as corresponds to call-time choice. In alternative derivations, $coin$ could be reduced to 1 (or to $\bot$). It is easy to see that $[\![heads(repeat(coin))]\!]^{\mathcal{P}}_{CRWL} = \{(0,0),(1,1),(\bot,0),(0,\bot),(\bot,1),(1,\bot),(\bot,\bot),\bot\}$.

Note that $(1,0),(0,1) \notin [\![heads(repeat(coin))]\!]^{\mathcal{P}}_{CRWL}$.

The following monotonicity lemma is a classical result in the *CRWL* framework [9, 10]:

LEMMA 1. *Given a program $\mathcal{P}$, $e \in Exp_\bot$, $t \in CTerm_\bot$ and $\theta, \theta' \in CSubst_\bot$ with $\theta \sqsubseteq \theta'$ then we have:*

$$\text{if } \mathcal{P} \vdash_{CRWL} e\theta \twoheadrightarrow t \text{ then } \mathcal{P} \vdash_{CRWL} e\theta' \twoheadrightarrow t$$

We stress the fact that the *CRWL*-calculus *is not* an operational mechanism for executing programs, but a way of describing the logic of programs. At the operational level the *CRWL* framework comes with various lazy narrowing-based goal-solving calculi [10, 26] not considered in this paper.

3. CRWL and rewriting: a first discussion

Our general concern is how to express non-strict call-time choice semantics by means of a simple rewriting-like one-step reduction relation. We started Section 1 by observing that ordinary term rewriting is not valid for that purpose. Now, we discard also the possibility of transforming the original system into another one such that using (ordinary) term rewriting it behaves as the original one under call-time choice. More precisely, we pose the following question:

> *For any given c-rewrite system $\mathcal{P}$, can we find another rewrite system (constructor based or not) $\mathcal{P}'$ such that for each expression e and constructed term t, (which can be ground or not) $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ iff $e \rightarrow^*_{\mathcal{P}'} t$?*

The answer to it is *'no'*, as the following simple example shows, exploiting the fact that rewriting is closed under substitutions while *CRWL*-provability is only closed under c-substitutions.

EXAMPLE 1. *Consider the rewrite system $\mathcal{P}$:*

$$f(X) \rightarrow c(X,X) \qquad coin \rightarrow 0 \qquad coin \rightarrow 1$$

and assume a system $\mathcal{P}'$ such that: $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \Leftrightarrow e \rightarrow^*_{\mathcal{P}'} t$, for all e, t. We will arrive to a contradiction.

Since $\mathcal{P} \vdash_{CRWL} f(X) \twoheadrightarrow c(X,X)$, we must have $f(X) \rightarrow^*_{\mathcal{P}'} c(X,X)$. Now, since $\rightarrow^*_{\mathcal{P}'}$ is closed under substitutions, we have $f(coin) \rightarrow^*_{\mathcal{P}'} c(coin, coin)$, and then we have the reductions $f(coin) \rightarrow^*_{\mathcal{P}'} c(coin,coin) \rightarrow^*_{\mathcal{P}'} c(0,1)$. But it is easy to see that $\mathcal{P} \vdash_{CRWL} f(coin) \twoheadrightarrow c(0,1)$ does not hold.

Another possibility is to impose the restriction that the substitution θ in a rewriting step must be a c-substitution, as it is done in the rule **(OR)** of *CRWL*. More precisely, we can define rewriting by the rule **(OR')** in Figure 4 below. With it the step $heads(repeat(coin)) \rightarrow heads(coin : repeat(coin))$ in the example of Figure 1 would not be legal anymore. This simple solution would be enough to deal with call-time choice and a *strict* semantics, but it is not sufficient for non-strictness, as shown by the following simple example:

EXAMPLE 2. *Consider the rewrite system given by the two rules $f(X) \rightarrow 0$ and $loop \rightarrow loop$. With a non-strict semantics $f(loop)$ should be reducible to 0. But with* **(OR')** *$f(loop) \rightarrow 0$ is not permitted; the only rewriting sequence starting with $f(loop)$ is $f(loop) \rightarrow f(loop) \rightarrow \ldots$, thus leaving $f(loop)$ semantically undefined, as would correspond to a strict semantics.*

What is missing is a rule allowing to reduce a not-needed (sub)-expression to a special constructor term with no information in it. Since not-neededness is undecidable, this special reduction must be allowed for any expression. This is given precisely by the rule **(B)** of *CRWL*, which is indeed a one-step rule. The result of this discussion is the one-step reduction relation $\rightarrowtail$ given in Figure 4.

It is not difficult to prove the following equivalence result:

THEOREM 1. *Let $\mathcal{P}$ be a CRWL-program, $e \in Exp_\bot, t \in CTerm_\bot$. Then $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ iff $e \rightarrowtail^*_{\mathcal{P}} t$.*

PROOF: It is easy to see that $\rightarrowtail^*$ (the reflexive and transitive closure of $\rightarrowtail$) coincides with the derivability relation defined by the proof calculus called *BRC* in [10]. This means that $\mathcal{P} \vdash_{BRC} e \twoheadrightarrow e'$ *iff* $e \rightarrowtail^* e'$. But in that paper it is proved that, for $e \in Exp_\bot$ and $t \in CTerm_\bot$, *BRC*-derivability and *CRWL*-derivability (called there *GORC*-derivability) are equivalent, what implies:
$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \Leftrightarrow \mathcal{P} \vdash_{BRC} e \twoheadrightarrow t \Leftrightarrow e \rightarrowtail^* t$. □

We remark that **(OR')** essentially corresponds to innermost evaluation. So the result has the following interesting reading: non-strict call-time choice can be achieved via innermost evaluation if at any step one has the possibility of reducing a subexpression to $\bot$. For instance, a $\rightarrowtail$-rewrite sequence with the example of Figure 1 would be:

$$\begin{array}{l} heads(repeat(coin)) \rightarrowtail heads(repeat(0)) \rightarrowtail \\ heads(0 : repeat(0)) \rightarrowtail heads(0 : 0 : repeat(0)) \rightarrowtail \\ heads(0 : 0 : \bot) \rightarrowtail (0,0) \end{array}$$

The rules for $\rightarrowtail$ can actually serve for a very easy implementation of non-strict call-time choice, but with a major drawback: reduction follows an unnatural order and requires, at any step, an unavoidable guessing between the two rules **(B')** and **(OR')**, leading to high inefficiency. Therefore, $\rightarrowtail$ achieves only partially our goals and we cannot consider it as the natural reduction notion we are looking for.

4. Rewriting with local bindings

In this section we introduce local bindings in the form of *let*-expressions as a convenient way of expressing sharing. Formally

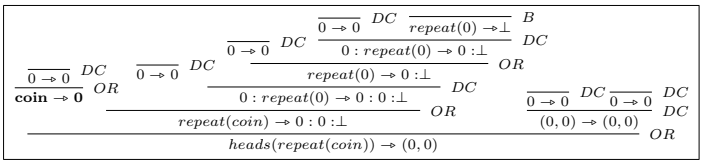


Figure 3. A *CRWL*-derivation

(B')	$\mathcal{C}[e]$	$\rightarrowtail$	$\mathcal{C}[\bot]$	for any $\mathcal{C} \in Cntxt, e \in Exp_\bot$
(OR')	$\mathcal{C}[f(t_1\theta,\ldots,t_n\theta)]$	$\rightarrowtail$	$\mathcal{C}[e\theta]$	for any $\mathcal{C} \in Cntxt$, $f(t_1,\ldots,t_n) \to e \in \mathcal{P}$, $\theta \in CSubst_\bot$

Figure 4. A one-step reduction relation for non-strict call-time choice

the syntax for *let-expressions* is:

$$LExp \ni e ::= X \mid h(\overline{e}) \mid let\ X = e_1\ in\ e_2$$

where $X \in \mathcal{V}$, $h \in CS \cup FS$, $\overline{e}$ is a tuple of *let*-expressions, and e_1, e_2 are single *let*-expressions. We will use the notation $let\ \overline{X = a}\ in\ e$ as a shortcut for $let\ X_1 = a_1\ in \ldots in\ let\ X_n = a_n\ in\ e$. The notion of *one-hole context* is also extended to the new syntax:

$$\mathcal{C} ::= [\,] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots,\mathcal{C},\ldots)$$

The sets $FV(e)$ of *free* and $BV(e)$ *bound* variables of $e \in LExp$ are defined as:

$$FV(X) = \{X\};\ FV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} FV(e_i);$$
$$FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\});$$
$$BV(X) = \emptyset;\ BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i);$$
$$BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\}$$

Notice that with the given definition of $FV(let\ X = e_1\ in\ e_2)$ there are not recursive *let*-bindings in the language since the possible occurrences of X in e_1 are not considered bound and therefore refer to a 'different' X. This is similar to what is done in [19], but not in [1, 14]. Recursive *let*s have their own interest but since they are not present in *CRWL*-programs (there are no *let*s at all in *CRWL*) and will neither appear in a *let*-rewriting reduction (to be defined below) unless they are already present in the c-rewrite system, we have decided not to consider them. Furthermore, there is not a general consensus about the reading of recursive *let*s in presence of non-determinism.

Notice that the notion of c-term has not changed with the introduction of *let*s: in particular c-terms do not contain *let*s, but can contain bound variables, as happens for example with (X, X) in the *let*-expression $let\ X = coin\ in\ (X, X)$.

As usual with syntactical binding constructs, we assume a variable convention according to what bound variables can be consistently renamed as to ensure that the same variable symbol does not occur free and bound within an expression. Moreover, to keep simple the management of substitutions, we assume that whenever θ is applied to an expression $e \in LExp$, the necessary renamings are done in e to ensure that $BV(e) \cap (dom(\theta) \cup ran(\theta)) = \emptyset$. With all these conditions the rules defining application of substitutions are simple while avoiding variable capture:

$$X\theta = \theta(X)$$
$$h(e_1,\ldots,e_n)\theta = h(e_1\theta,\ldots,e_n\theta)$$
$$(let\ X = e_1\ in\ e_2)\theta = let\ X = e_1\theta\ in\ e_2\theta$$

The *let*-rewriting relation $\rightarrow_l$ is shown in Figure 5. The rule **(Fapp)** performs a rewriting step in a proper sense, using a rule of the program. Note that only c-substitutions are allowed, to avoid copying of unevaluated expressions which would destroy sharing and call-time choice. **(Contx)** allows to select any subexpression as a redex for the derivation. The rest of the rules are syntactic manipulations of *let*-expressions. In particular **(LetIn)** transforms standard expressions by introducing a *let*-binding to express sharing. On the other hand, **(Bind)** removes a *let*-construction for a variable when its binding expression has been evaluated. **(Elim)** allows to remove a binding when the variable does not appear in the body of the construction, which means that the corresponding value is not needed for evaluation. This rule is needed because the expected normal forms are c-terms not containing *let*s. **(Flat)** is needed for flattening nested *let*s, otherwise some reductions could become wrongly blocked or forced to diverge. For example, with the rewrite rules $loop \to loop$ and $g(s(X)) \to 1$ and applying twice **(LetIn)** to the expression $g(s(loop))$, we obtain $let\ X = (let\ Y = loop\ in\ s(Y))\ in\ g(X)$. Without **(Flat)** we can only perform reductions on $loop$; with **(Flat)** we obtain $let\ Y = loop\ in\ let\ X = s(Y)\ in\ g(X)$ and then applying **(Bind)** and **(Elim)** we achieve the expected value 1. Notice that with the variable convention, the condition $Y \notin FV(e_3)$ in **(Flat)** would not be needed. We have written it in order to keep the rules independent of the convention. Quite different is the case of **(Elim)**, where the condition $X \notin FV(e_2)$ might hold or not.

As a complete derivation example, consider the program of Figure 1 and the derivation of Figure 6. Notice that there is not a unique $\rightarrow_l$-reduction leading to $(0, 0)$. The definition of $\rightarrow_l$ does not prescribe any particular strategy, a subject that has been left out of the scope of this paper.

5. Equivalence of let-rewriting and CRWL

In this section we will prove the soundness and completeness results of *let*-rewriting with respect to *CRWL*. To this purpose we will need to consider $\bot$ at some points. Therefore we define the set $LExp_\bot$ in the natural way. We also define the *shell* $|e|$ *of an expression* e that represents the outer constructor part of e, and is defined as follows:

$$\begin{array}{lcl} |X| & = & X \\ |c(e_1,\ldots,e_n)| & = & c(|e_1|,\ldots,|e_n|) \\ |f(e_1,\ldots,e_n)| & = & \bot \\ |let\ X = e_1\ in\ e_2| & = & |e_2|[X/|e_1|] \end{array}$$

(Contx)	$\mathcal{C}[e] \to_l \mathcal{C}[e'], \quad$ if $e \to_l e', \mathcal{C} \in Cntxt$
(LetIn)	$h(\ldots, e, \ldots) \to_l let\ X = e\ in\ h(\ldots, X, \ldots)$ if $h \in CS \cup FS$, e takes one of the forms $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable
(Flat)	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \ \to_l \ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ assuming that Y does not appear free in e_3
(Bind)	$let\ X = t\ in\ e \ \to_l \ e[X/t], \quad$ if $t \in CTerm$
(Elim)	$let\ X = e_1\ in\ e_2 \to_l e_2, \quad$ if X does not appear free in e_2
(Fapp)	$f(t_1\theta, \ldots, t_n\theta) \ \to_l \ e\theta, \quad$ if $f(t_1, \ldots, t_n) \to e \in \mathcal{P}, \theta \in CSubst$

Figure 5. Rules of *let*-rewriting

$heads(repeat(coin)) \to_l$	(**LetIn**)
$let\ X = repeat(coin)\ in\ heads(X) \to_l$	(**LetIn**)
$let\ X = (let\ Y = coin\ in\ repeat(Y))\ in\ heads(X) \to_l$	(**Flat**)
$let\ Y = coin\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	(**Fapp**)
$let\ Y = 0\ in\ let\ X = repeat(Y)\ in\ heads(X) \to_l$	(**Bind**)
$let\ X = repeat(0)\ in\ heads(X) \to_l$	(**Fapp**)
$let\ X = 0 : repeat(0)\ in\ heads(X) \to_l$	(**LetIn**)
$let\ X = (let\ Z = repeat(0)\ in\ 0 : Z)\ in\ heads(X) \to_l$	(**Flat**)
$let\ Z = repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**Fapp**)
$let\ Z = 0 : repeat(0)\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**LetIn**, **Flat**)
$let\ U = repeat(0)\ in\ let\ Z = 0 : U\ in\ let\ X = 0 : Z\ in\ heads(X) \to_l$	(**Bind**), **2**
$let\ U = repeat(0)\ in\ heads(0 : 0 : U) \to_l$	(**Fapp**)
$let\ U = repeat(0)\ in\ (0, 0) \to_l$	(**Elim**)
$(0, 0)$	

Figure 6. A let-rewriting derivation

Notice that the information contained in *let*-bindings is taken into account for building up the shell of an expression.

5.1 Soundness

Concerning soundness we would like to prove something like this:

If $e \to_l e'$ *then* $[\![e']\!]_{CRWL} \subseteq [\![e]\!]_{CRWL}$, for any $e, e' \in Exp$.

That is, $\to_l$-steps do not create new *CRWL*-semantic values. But *let*-expressions are not defined in *CRWL* and even if we start with an expression without *let*s, *let*-rewriting may introduce them by **(LetIn)**. To cope with this situation we enlarge the *CRWL*-calculus in Figure 2 to a new calculus $CRWL_{let}$, by adding a new rule for dealing with *let*-expressions:

$$\textbf{(Let)} \ \frac{e_1 \twoheadrightarrow t_1 \quad e[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e \twoheadrightarrow t}$$

We write $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$ if $e \twoheadrightarrow t$ is derivable in the $CRWL_{let}$ calculus using the program $\mathcal{P}$. The $CRWL_{let}$-denotation of an expression $e \in LExp_\bot$ with respect to the program $\mathcal{P}$ is defined as

$$[\![e]\!]^{\mathcal{P}}_{CRWL_{let}} = \{t \in CTerm_\bot | \mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t\}$$

We will omit the sub(super)-scripts when they are clear by the context.

$CRWL_{let}$ shares with *CRWL* the property of closedness under c-substitutions. The following result states this fact, together with some other useful properties related to shells that are not difficult to check by the appropriate induction in each case:

LEMMA 2. *Let $\mathcal{P}$ be a CRWL-program and $e \in LExp_\bot$. Then:*
(i) $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$ implies $\mathcal{P} \vdash_{CRWL_{let}} e\sigma \twoheadrightarrow t\sigma$, for any $t \in CTerm_\bot, \sigma \in CSubst_\bot$.
(ii) $|e| \in [\![e]\!]_{CRWL_{let}}$.
(iii) $[\![e]\!]_{CRWL_{let}} \subseteq (|e|\uparrow)\downarrow$, where for a given $E \subseteq LExp_\bot$ its upward closure is $E\uparrow = \{e' \in LExp_\bot | \ \exists e \in E.\ e \sqsubseteq e'\}$, its downward closure is $E\downarrow = \{e' \in LExp_\bot | \ \exists e \in E.\ e' \sqsubseteq e\}$, and those operators are overloaded for let-expressions as $e\uparrow = \{e\}\uparrow$ and $e\downarrow = \{e\}\downarrow$ [1].
(iv) $e \to_l e'$ implies $|e| \sqsubseteq |e'|$.

Parts *(ii)* to *(iv)* express that the shell of an expression represents 'stable' information contained in the expression (*(ii)* says that shells are in the denotation; *(iii)*, that everything in the denotation comes from refining it, and *(iv)* says that shells grow monotonically with reduction).

It is easy to establish the equivalence between *CRWL* and $CRWL_{let}$ for expressions not involving *let*s.

LEMMA 3. *For any CRWL-program $\mathcal{P}$, $e \in Exp_\bot$ and $t \in CTerm_\bot$, we have: $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ iff $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$. Therefore $[\![e]\!]^{\mathcal{P}}_{CRWL} = [\![e]\!]^{\mathcal{P}}_{CRWL_{let}}$.*

With the aid of $CRWL_{let}$, the theorem we are looking for can be stated as follows:

THEOREM 2 (One-Step Soundness of *let*-rewriting).
For any $e, e' \in LExp$,

$$e \to_l e' \textit{ implies } [\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}.$$

Notice that because of non-determinism $\subseteq$ cannot be replaced by $=$ in this theorem. The proof of Theorem 2 (which is given below) would proceed straightforwardly by a case distinction on

[1] There is a mistake in the original formulation of this result from [16], where the additional downward closure is missing.

the rules for $\rightarrow_l$, if the following *monotonocity under contexts* was true for any context $\mathcal{C}$:

$$[\![e]\!]_{CRWL_{let}} \subseteq [\![e']\!]_{CRWL_{let}} \textit{ implies } [\![\mathcal{C}[e]]\!]_{CRWL_{let}} \subseteq [\![\mathcal{C}[e']]\!]_{CRWL_{let}}$$

Unfortunately this property is false because of the possible capture of variables when switching from e to $\mathcal{C}[e]$, as the following example shows:

EXAMPLE 3. *If f is defined by $f(0) \rightarrow 1$ we have*

$$\{\bot\} \equiv [\![f(X)]\!] \subseteq [\![0]\!] \equiv \{\bot, 0\}$$

but when these expressions are placed within the context $let\ X = 0\ in\ [\,]$ we obtain

$$\{\bot, 1\} \equiv [\![let\ X = 0\ in\ f(X)]\!] \not\subseteq [\![let\ X = 0\ in\ 0]\!] \equiv \{\bot, 0\}.$$

To overcome this problem and prove Theorem 2 we need a stronger result showing that $\rightarrow_l$-steps preserve (in the sense of $\subseteq$) the $CRWL_{let}$-semantics even under substitutions. To formalize the idea some new notions are useful:

DEFINITION 1 (Hypersemantics).

(i) The hypersemantics *of an expression $e \in LExp_\bot$, written as $[\![e]\!]_{CRWL_{let}}$, is a mapping from $CSubst_\bot$ into $\mathcal{P}(CTerm_\bot)$ defined as*

$$[\![e]\!]_{CRWL_{let}}\ \theta = [\![e\theta]\!]_{CRWL_{let}}.$$

(ii) Hypersemantics of expressions are ordered as follows:

$$[\![e_1]\!]_{CRWL_{let}} \Subset [\![e_2]\!]_{CRWL_{let}} \textit{ iff } [\![e_1\theta]\!]_{CRWL_{let}} \subseteq [\![e_2\theta]\!]_{CRWL_{let}},\ \forall \theta \in CSubst_\bot$$

In other terms, iff $\forall \theta \in CSubst_\bot$, $\mathcal{P} \vdash_{CRWL_{let}} e_1\theta \twoheadrightarrow t$ implies $\mathcal{P} \vdash_{CRWL_{let}} e_2\theta \twoheadrightarrow t$.

Hypersemantics fulfils the desired monotonicity property:

LEMMA 4. *For any $e, e' \in LExp_\bot$, and every context $\mathcal{C}$ we have:*

$$[\![e]\!]_{CRWL_{let}} \Subset [\![e']\!]_{CRWL_{let}} \textit{ implies } [\![\mathcal{C}[e]]\!]_{CRWL_{let}} \Subset [\![\mathcal{C}[e']]\!]_{CRWL_{let}}$$

Now the idea is to prove for hypersemantics a result analogous to Theorem 2, which will become then an easy corollary. Two more lemmas are needed: the first is a standard substitution lemma and the second is a classical result for *CRWL* [10], that is also valid for $CRWL_{let}$.

LEMMA 5. *Given $e, e' \in LExp_\bot$, $\theta \in Subst_\bot$ and $X \in \mathcal{V}$ such that $X \notin dom(\theta)$ and $X \notin ran(\theta)$, then we have $(e[X/e'])\theta \equiv e\theta[X/e'\theta]$.*

LEMMA 6. *Let $e, e' \in LExp_\bot$, $t, t' \in CTerm_\bot$ be such that $e \sqsubseteq e'$ and $t \sqsupseteq t'$. If $e \twoheadrightarrow t$ then $e' \twoheadrightarrow t'$ with a proof of the same size or smaller.*

All these results allow to prove the expected generalization of Theorem 2 to hypersemantics.

THEOREM 3 (One-Step Hyper-Soundness of *let*-rewriting).
For any $e, e' \in LExp$

$$e \rightarrow_l e' \textit{ implies } [\![e']\!]_{CRWL_{let}} \Subset [\![e]\!]_{CRWL_{let}}$$

And now Theorem 2 follows naturally:
PROOF:[For Theorem 2] Assume $e \rightarrow_l e'$. By Theorem 3 we have $[\![e']\!]_{CRWL_{let}} \Subset [\![e]\!]_{CRWL_{let}}$, and therefore $[\![e'\theta]\!]_{CRWL_{let}} \subseteq [\![e\theta]\!]_{CRWL_{let}}$ for each $\theta \in CSubst_\bot$. Choosing $\theta = \epsilon$ (the empty substitution) we obtain $[\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}$ as desired. □

One-step soundness as given by Theorem 2 is straightforwardly extended to several steps, that is, to the transitive and reflexive closure $\rightarrow_l^*$ of the *let*-rewriting relation $\rightarrow_l$:

COROLLARY 1. *For any $e, e' \in LExp$*

$$e \rightarrow_l^* e' \textit{ implies } [\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}$$

PROOF: An immediate induction on the length of the derivation $e \rightarrow_l^* e'$. □

Finally we can easily get our main result concerning the soundness of *let*-rewriting with respect not only to the $CRWL_{let}$ calculus, but also to the original *CRWL* formulation:

THEOREM 4 (Soundness of *let*-rewriting).
Let $\mathcal{P}$ be a CRWL-program and $e \in Exp$. Then:
(i) $e \rightarrow_l^* e'$ implies $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow |e'|$, for any $e' \in LExp$.
(ii) $e \rightarrow_l^* t$ implies $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, for any $t \in CTerm$.

PROOF: *(i)*: Assume $e \rightarrow_l^* e'$. Then, by Corollary 1 we have $[\![e']\!]_{CRWL_{let}} \subseteq [\![e]\!]_{CRWL_{let}}$. Since $|e'| \in [\![e']\!]_{CRWL_{let}}$ by Lemma 2, we get $|e'| \in [\![e]\!]_{CRWL_{let}}$, which means $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow |e'|$. By Lemma 3, we conclude $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow |e'|$.
(ii): trivial by *(i)*, since $|t| = t$ for $t \in CTerm$. □

5.2 Completeness

Now we look for the reverse implication of Theorem 4. Some additional results are needed for it. The first one concerns only $\rightarrow_l$-reductions:

LEMMA 7 (Peeling lemma). *For any $e \in LExp$ we have that*

$$e \rightarrow_l^* let\ \overline{X = a}\ in\ b$$

for some $\overline{a} \subseteq LExp$ such that $|a_i| = \bot$ for all $a_i \in \overline{a}$, and some $b \in LExp$ such that either $b \in \mathcal{V}$ or $b \equiv g(\overline{t})$ with $g \in CS \cup FS, \overline{t} \subseteq CTerm$.
Moreover, if $e \equiv h(e_1, \ldots, e_n)$ with $h \in CS \cup FS$, then

$$e \equiv h(e_1, \ldots, e_n) \rightarrow_l^* let\ \overline{X = a}\ in\ h(t_1, \ldots, t_n)$$

under the conditions above, and verifying also that $t_i \equiv e_i$ whenever $e_i \in CTerm$.
Besides, we can state that in these derivations the rule **(Fapp)** *was not applied.*[2]

We can think about a *let*-expression as a regular $CRWL$-term in which some additional sharing information has been encoded using *let* expressions. As we do not use the rule **(Fapp)** in the derivations for this lemma, we do not make progress in the evaluation of the implicit $CRWL$-term corresponding to e (thus not changing the corresponding $CRWL$-denotation), but we change the sharing-enriched representation of this $CRWL$-term in the *let*-rewriting syntax. What we do in these derivations is exposing the computed part of e concentrating it in $g(\overline{t})$, that is, the part whose shell is different from $\bot$. That is why we call it '*Peeling lemma*'.

The next result is already a technical completeness result preparing for our completeness theorems below:

LEMMA 8. *Let $\mathcal{P}$ be a CRWL-program, $e \in Exp$, and $t \in CTerm_\bot$ such that $t \neq \bot$. Then:*

[2] The formulation of this Lemma is slightly different from the original version of [16], which was wrong and only worked for *let*-expressions of the shape $h(\overline{e})$—for example consider $e \equiv let\ X = coin\ in\ X$. Those problems have been fixed in this new version, but anyway the impact of this erratum in the rest of the work is negligible, as it this Lemma is only used in the proof for Lemma 8, where it is applied to expressions of the shape $h(\overline{e})$.

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \textit{ implies } e \to_l^* \textit{let } \overline{X = a} \textit{ in } t'$$

for some $t' \in CTerm$ *and* $\overline{a} \subseteq LExp$ *in such a way that* $t \sqsubseteq |\textit{let } \overline{X = a} \textit{ in } t'|$ *and* $|a_i| = \bot$ *for all* $a_i \in \overline{a}$. *As a consequence,* $t \sqsubseteq t'[\overline{X}/\bot]$.

Our main results concerning the completeness of *let*-rewriting are now easy consequences of Lemma 8. The first shows that any c-term obtained by *CRWL* for an expression can be refined by a *let*-rewriting derivation.

THEOREM 5 (Completeness of *let*-rewriting).
Let $\mathcal{P}$ *be a CRWL-program,* $e \in Exp$, *and* $t \in CTerm_\bot$. *Then:*

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \textit{ implies } e \to_l^* e'$$

for some $e' \in LExp$ *such that* $t \sqsubseteq |e'|$.

PROOF: If $t = \bot$ then we are done with $e \to_l^0 e$ as $\forall e, \bot \sqsubseteq |e|$. If $t \neq \bot$ then by Lemma 8 we have $e \to_l^* \textit{let } \overline{X} = \overline{a} \textit{ in } t'$ such that $t \sqsubseteq |\textit{let } \overline{X} = \overline{a} \textit{ in } t'|$. □

The next result considers the case of total c-terms:

THEOREM 6 (Completeness of *let*-rewriting for total solutions).
Let $\mathcal{P}$ *be a CRWL-program,* $e \in Exp$, *and* $t \in CTerm$. *Then:*

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \textit{ implies } e \to_l^* t.$$

PROOF: Assume $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, then by Lemma 8 we get $e \to_l^* \textit{let } \overline{X} = \overline{a} \textit{ in } t'$ such that $t \sqsubseteq |\textit{let } \overline{X} = \overline{a} \textit{ in } t| \equiv t'[\overline{X}/\bot]$, for some $t' \in CTerm, \overline{a} \subseteq LExp$. As $t \in CTerm$ then t is maximal w.r.t. $\sqsubseteq$, so $t \sqsubseteq t'[\overline{X}/\bot]$ implies $t'[\overline{X}/\bot] \equiv t$, but then $t'[\overline{X}/\bot] \in CTerm$ so it must happen that $FV(t') \cap \overline{X} = \emptyset$ and therefore $t' \equiv t'[\overline{X}/\bot] \equiv t$. But then $\textit{let } \overline{X} = \overline{a} \textit{ in } t' \to_l^* t' \equiv t$ by zero or more steps of **(Elim)**, so $e \to_l^* \textit{let } \overline{X} = \overline{a} \textit{ in } t' \to_l^* t$, that is $e \to_l^* t$. □

As a final corollary of this result and the part *(ii)* of the soundness Theorem 4 we obtain a strong equivalence result for both formalisms:

THEOREM 7 (Equivalence of *CRWL* and *let*-rewriting).
Let $\mathcal{P}$ *be a CRWL-program,* $e \in Exp$, *and* $t \in CTerm$. *Then:*

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \textit{ iff } e \to_l^* t.$$

This constitutes the main result in the paper.

6. Let-rewriting versus classical rewriting

In this section we examine the relationship between *let*-rewriting and ordinary rewriting for TRS. We will first prove in 6.1 that *let*-rewriting is sound with respect to rewriting. As we know since the discussion starting the paper, completeness does not hold in general because, in presence on non-determinism, rewriting (that corresponds to run-time choice) can obtain more results than *let*-rewriting (call-time choice). However, we will be able to prove completeness for programs that are *deterministic*, a property close to confluence that will be defined in 6.2.

Thanks to the equivalence of *CRWL* and *let*-rewriting we can choose the most appropriate point of view for each of the two goals (soundness and completeness): we will use *let*-rewriting for proving soundness, and the proof calculus of *CRWL* for defining the property of determinism and proving that, under determinism, completeness holds.

6.1 Soundness of let-rewriting w.r.t. classical rewriting

Firstly, we need a syntactic transformation from $LExp$ into Exp, removing the *let* constructions (thus losing the sharing information they provide). Given $e \in LExp$ we define its transformation into a standard expression $\widehat{e}$ as:

$$\widehat{X} \equiv X$$
$$\widehat{h(e_1, \ldots, e_n)} \equiv h(\widehat{e_1}, \ldots, \widehat{e_n})$$
$$\widehat{\textit{let } X_p = e_1 \textit{ in } e_2} \equiv \widehat{e_2}[X_p/\widehat{e_1}]$$

This transformation satisfies the following properties, which can be proved by induction on the structure of *let*-expressions:

LEMMA 9. *For all* $e \in LExp$ *we have* $\widehat{e} \in Exp$, $var(\widehat{e}) \subseteq FV(e)$, $|\widehat{e}| \equiv |e|$. *Moreover, for all* $e \in Exp$ *we have* $\widehat{e} \equiv e$.

The following lemmas can be easily proved by induction on the structure of expressions:

LEMMA 10. *For all* $e, s, s' \in Exp$, $X \in \mathcal{V}$, $s \to^* s'$ *implies* $e[X/s] \to^* e[X/s']$.

LEMMA 11. *For all* $e, s \in LExp$, $X \in \mathcal{V}$: $\widehat{e[X/s]} \equiv \widehat{e}[X/\widehat{s}]$.

Using these lemmas we get a first soundness result, stating that what can be done in one step of *let*-rewriting, can also be done in zero or more steps of ordinary rewriting, after erasing the sharing information by the transformation $\widehat{\ }$:

LEMMA 12. *For all* $e, e' \in LExp$ *we have:* $e \to_l e'$ *implies* $\widehat{e} \to^* \widehat{e'}$.

Some other soundness results follow easily from the lemma above. The first one expresses that any expression (not involving *let*'s) reachable by *let*-rewriting can be also reached by ordinary rewriting. In other terms, *let*-rewriting ($\to_l^*$) is a sub-relation of rewriting ($\to^*$), when ($\to_l^*$) is restricted to ordinary expressions (not involving *let*'s).

THEOREM 8.
For any $e, e' \in LExp$, $e \to_l^* e'$ *implies* $\widehat{e} \to^* \widehat{e'}$. *As a consequence, if* $e, e' \in Exp$, *then* $e \to_l^* e'$ *implies* $e \to^* e'$.

PROOF: An immediate induction on the length of the *let*-derivation, using Lemma 12 for the inductive step. For the remaining statement, if $e, e' \in Exp$ then $e \equiv \widehat{e}, e' \equiv \widehat{e'}$ by Lemma 9, and therefore $e \equiv \widehat{e} \to^* \widehat{t} \equiv t$. □

The next result, based on the correspondence of *CRWL* and *let*-rewriting established in Section 5, is a soundness theorem for *CRWL* with respect to ordinary rewriting.

THEOREM 9. *For all* $e \in Exp$ *and* $t \in CTerm_\bot$, $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ *implies* $\exists e' \in Exp$ *such that* $e \to^* e'$ *and* $t \sqsubseteq |e'|$.

PROOF: Assume $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, then by Theorem 5 $\exists e'' \in LExp$ such that $e \to_l^* e''$ and $t \sqsubseteq |e''|$. Then by Theorem 8 combined with Lemma 9 we get $e \equiv \widehat{e} \to^* \widehat{e''}$. But then we can choose $e' \equiv \widehat{e''}$ because $\widehat{e''} \in Exp$ by Lemma 9, and $|e'| \equiv |\widehat{e''}| = |e''| \sqsupseteq t$, by Lemma 9 again. □

6.2 Completeness of *CRWL* w.r.t. classical rewriting

As commented before, we cannot expect to get a completeness result of the *CRWL* framework w.r.t. classical rewriting for any program, but only for the class of deterministic programs, which are defined as follows:

DEFINITION 2 (Deterministic *CRWL*-program).
A CRWL-program $\mathcal{P}$ *is* deterministic *iff the denotation* $[\![e]\!]^{\mathcal{P}}$ *of any expression* $e \in Exp_\bot$ *is a directed set. In other words, iff* $\forall e \in Exp_\bot$ *and* $t_1, t_2 \in [\![e]\!]^{\mathcal{P}}$ *there exists* $t_3 \in [\![e]\!]^{\mathcal{P}}$ *with* $t_1 \sqsubseteq t_3$ *and* $t_2 \sqsubseteq t_3$.

Determinism as defined here is intuitively close to confluence, but the two notions do not coincide. Determinism does not imply confluence, as the following example shows:

EXAMPLE 4. *Consider the program $\mathcal{P}$ given by the three rules*

$$f \to a \qquad f \to loop \qquad loop \to loop$$

where a is a constructor. It is clear that $\mathcal{P}$ is not confluent (f can be reduced to a and loop, which cannot be joined to a common reduct), but is deterministic, since $[\![f]\!]^{\mathcal{P}} = \{\bot, a\}$, $[\![loop]\!]^{\mathcal{P}} = \{\bot\}$ and $[\![a]\!]^{\mathcal{P}} = \{\bot, a\}$, each of them being a directed set.

We conjecture that the reverse implication (confluence $\Rightarrow$ determinism) is true, but a precise proof of this fact seems surprisingly complicated and we have not yet completed it.

Determinism has been defined as a semantic property. However, thanks to the equivalence of *CRWL* and *let*-rewriting, it can be also characterized in terms of reduction, as the following result shows:

LEMMA 13. *A CRWL-program $\mathcal{P}$ is deterministic iff for any expressions $e \in Exp$, $e', e'' \in LExp$ with $e \to_l^* e'$ and $e \to_l^* e''$, there exists $e''' \in LExp$ such that $e \to_l^* e'''$ and $|e'''| \sqsupseteq |e'|, |e'''| \sqsupseteq |e''|$*[3].

We do not know if in this result *let*-rewriting can be replaced by ordinary rewriting.

We need also the following auxiliary notions:

DEFINITION 3 (Denotation for a substitution).
Given a CRWL-program P, for all $\sigma \in Subst_\bot$ its denotation is defined as $[\![\sigma]\!] = \{\theta \in CSubst_\bot | \forall X \in \mathcal{V},\ \mathcal{P} \vdash_{CRWL} \sigma(X) \twoheadrightarrow \theta(X)\}$[4].

DEFINITION 4 (Deterministic substitution).
The set of deterministic substitutions *for a given CRWL-program $\mathcal{P}$, $DSubst_\bot$ is defined as*

$$DSubst_\bot = \{\theta \in Subst_\bot |\quad \forall X \in dom(\theta).\ [\![\theta(X)]\!]^{\mathcal{P}} \text{ is a directed set}\}$$

Using these notions we can develop an extension of the proof calculus for *CRWL* which does call-by-name parameter passing only when it is safe for call-time choice. The extended calculus $CRWL^d$ contains the same rules of *CRWL* and the following additional rule:

$$(\mathbf{OR^d})\ \frac{r\theta \twoheadrightarrow t}{f(\overline{p})\theta \twoheadrightarrow t} \quad \text{if } (f(\overline{p}) \to r) \in \mathcal{P} \text{ and } \theta \in DSubst_\bot$$

Besides, for every $e \in Exp_\bot$ we define its denotation in this calculus as $[\![e]\!]^d = \{t \in CTerm_\bot | \mathcal{P} \vdash_{CRWL^d} e \twoheadrightarrow t\}$. Notice that this relation is undecidable (as happens with confuence) because the problem of checking whether a *CRWL*-denotation is a directed set or not is undecidable.

We will see that $CRWL^d$ proves exactly the same approximation statements that *CRWL* proves; to do that we must prove first the following auxiliary results:

LEMMA 14. *For any CRWL-program $\mathcal{P}$ and for all $\sigma \in DSusbt_\bot$, $[\![\sigma]\!]$ is a directed set.*

LEMMA 15. *For any CRWL-program $\mathcal{P}$ and for all $\sigma \in DSusbt_\bot$, $e \in Exp_\bot, t \in CTerm_\bot$, $\mathcal{P} \vdash_{CRWL} e\sigma \twoheadrightarrow t$ implies $\exists \theta \in [\![\sigma]\!]$ such that $\mathcal{P} \vdash_{CRWL} e\theta \twoheadrightarrow t$.*

[3] There is a typo in the original formulation of this Lemma from [16], where it says that $e', e'', e'' \in Exp$ instead of $e', e'', e'' \in LExp$.

[4] The definition of $[\![\sigma]\!]$ originally presented in [16] included an additional condition over $\theta \in [\![\sigma]\!]$ requiring that $dom(\theta) = dom(\sigma)$. With that definition Lemma 15 is false, so that condition had to be dropped in order to fix the proof for that Lemma.

Now we have at our disposal the tools needed to state and prove the adequacy of $CRWL^d$:

THEOREM 10. *For any CRWL-program $\mathcal{P}$ and $\forall e \in Exp_\bot$, $[\![e]\!]^d = [\![e]\!]$.*

Now we are ready to prove our first completeness result:

LEMMA 16. *For any CRWL-program $\mathcal{P}$, if it is deterministic then for all $e, e' \in Exp$, $e \to^* e'$ implies $[\![e']\!] \subseteq [\![e]\!]$.*

The previous lemma, together with the equivalence of *CRWL* and *let*-rewriting given by Theorem 7, allows to obtain strong relationships between rewriting, let-rewriting and *CRWL*, for the class of determinsitic programs.

THEOREM 11.
Let $\mathcal{P}$ be a deterministic CRWL-program, $e, e' \in Exp, t \in CTerm$. Then:
a) $e \to^* e'$ implies $e \to_l^* e''$ for some $e'' \in LExp$ with $|e''| \sqsupseteq |e'|$.
b) $e \to^* t$ iff $e \to_l^* t$ iff $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$.

PROOF: *a)* Assume $e \to^* e'$. Then $[\![e']\!] \subseteq [\![e]\!]$ by Lemma 16. Now, it is a known property of *CRWL* that $|e'| \in [\![e']\!]$, and then $|e'| \in [\![e]\!]$, which means that $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow |e'|$. Therefore, by Theorem 7 there exists $e'' \in LExp$ such that $e \to_l^* e''$ with $|e''| \sqsupseteq |e'|$.

b) That $e \to_l^* t$ iff $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, and that $e \to_l^* t$ implies $e \to^* t$ have been already proved for arbitrary programs in Theorems 7 and 8 respectively. What remains to be proved is that $e \to^* t$ implies $e \to_l^* t$ (i.e., $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$). Assume $e \to^* t$. Then $[\![t]\!] \subseteq [\![e]\!]$ by Lemma 16. Now, it is an easy property of *CRWL* that $t \in [\![t]\!]$, and therefore $t \in [\![e]\!]$, which exactly means that $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$. □

Notice that in part a) we cannot ensure $e \to^* e'$ implies $e \to_l^* e'$, because rewriting can reach some intermediate expressions not reachable by *let*-rewriting. For instance, given the deterministic program with the rules $g \to a$ and $f(x) \to c(x, x)$, we have $f(g) \to^* c(g, a)$, but not $f(g) \to_l^* c(g, a)$. Still, part a) is a strong completeness result for let-rewriting wrt rewriting for deterministic programs, since it says that the outer constructed part obtained in a rewriting derivation can be also obtained or even refined in a *let*-derivation. Combined with Theorem 8, part a) expresses a kind of equivalence between *let*-rewtiting and rewriting, valid for general derivations, even non-terminating ones. For terminated derivations reaching a constructor term (not further reducible), part b) gives an even stronger equivalence result.

7. Related work and conclusions

This work tries to fill a gap existing in the functional logic programming field, which is the technical disconnection between the two most accepted approaches to the paradigm: one, given by the *CRWL* framework, more biased to the semantics, and the other, focused in operational aspects, based on the theory or term rewriting. We feel that the missing piece was a precise, simple, high level and clear one-step reduction mechanism that is close to rewriting but at the same time respects call-time choice semantics for possibly non-confluent and non-terminating constructor-based rewrite systems.

There exist previous proposals that combine sharing with rewriting or narrowing, even for the specific case of functional logic programs. We briefly discuss now why we decided not to adopt them for our aim of comparison with *CRWL*.

A usual approach to expressing different levels of sharing in rewriting is term graph rewriting [21], a variant of which for constructor based systems was studied in [6, 7]. However, the class of programs is smaller in that work, since rewrite rules in term graph

rewrite systems must be orthogonal and extra variables are not considered. These restrictions were dropped in [3], but it does not contain any formal treatment for the properties of the proposed notions. Furthermore, and admitting that this is arguable, we consider that graph rewriting is a complex mechanism to reason about. For instance, we see graph homomorphisms as a more involved notion than matching. Therefore, we find it more comfortable, whenever possible, to use textual or equational counterparts of graph rewriting, as in essence is our *let*-rewriting or the λ-calculus with sharing of [19].

In [1] there is a proposal of two operational (natural and small-step) semantics for functional logic programs supporting sharing (call-time choice semantics), using a flat representation of programs coming from an implicit program transformation encoding the demand analysis used by needed narrowing, and some kind of heaps to express bindings for variables. As in our case, *let*-expressions are used to express sharing. The approach is useful as a technical basis for implementation and program manipulation purposes; but we think that, as happens with *CRWL* but for rather different reasons –too low-level and close to a particular operational strategy– it cannot be seen as the 'essential' basic reduction mechanism to understand non-strict call-time choice. Furthermore, to relate technically *CRWL* with [1] turns out to be a really hard task, that has been done in [15] but only for a restricted class of programs and expressions.

Local bindings *let X=e in* ... resemble oriented conditions $e \rightarrow X$ of the deterministic conditional rewrite systems of [20]. But they consider 3-CTRS systems and, most importantly, a different semantics for equality, according to which call-time choice is not respected.

Finally, for proving the completeness of a transformation that eliminates extra variables, [5] uses a variant or rewriting explicit substitution. However, their variant performs sharing only for the extra variables to be eliminated and not for the whole process of rewriting, and then they do not really achieve call-time choice.

Our concrete contributions can be summarized as follows:

- We have further clarified the well known fact that ordinary rewriting is not adequate for call-time choice, by showing that no program transformation can serve to fully simulate call-time choice by ordinary rewriting (Sect. 3). Therefore, the classical theory of TRS cannot serve as technical foundation for functional logic programs with call-time choice. Then we have proposed two one-step variants of rewriting.
- The first variant (Sect. 3) is very simple but of limited interest since it alters the natural sequence of rewriting in real computations.
- The second one (called *let*-rewriting in the paper) defines rewriting with local bindings. The rules for *let*-rewriting are very similar, but adapted to term rewriting with call-time choice, to those for λ-calculus with sharing [19], and can be seen as a particular textual (equational) presentation of graph term rewriting [21].
- As a major technical task we have proved the equivalence of *let*-rewriting and *CRWL*, which is the core of our contribution. Equivalence is a strong result that allows to apply known and future results about *CRWL* to *let*-rewriting and viceversa. Just to mention an example, the program transformations proved to be correct for *CRWL* in [15] are also valid for *let*-rewriting. As a technical tool for proving equivalence we have extended the *CRWL* logic itself to deal with local bindings, which might be a useful side-product.
- We have proved that for deterministic programs (a semantic condition very close to confluence) let-rewriting (hence *CRWL*-derivability) and ordinary rewriting coincide in some precise technical sense, while in general let-rewriting is a sub-relation of rewriting. We stress the fact that this is a new, technically non-trivial result connecting the *CRWL* and rewrite worlds; to the best of our knowledge, this kind of results were completely missing in the literature. Furthermore, we strongly conjecture (and we are hopefully very close to a complete proof) that confluence of a *CRWL*-program (in the ordinary sense of TRS) implies semantic determinism, which will imply that under confluence rewriting and let-rewriting are equivalent in some technical sense. This very intuitive (but hard to prove!) result will give further evidence (if it finally becomes proved) of the benefits of having connected *CRWL* and rewriting, since a result related purely to rewriting would become proved using semantical reasoning tools.

We must warn that *let*-rewriting as presented in this paper does not pretend to be in its own the working operational procedure for c-rewrite systems with call-time choice (functional-logic programs), for several reasons: first, we have not considered any rewriting strategy – something needed in practice – otherwise the rewriting space is too large. Second, there are two situations in computations where rewriting is not enough and must be lifted to narrowing: when the program uses extra variables (narrowing must be used then to obtain their values; rewriting 'magically' guesses them in the parameter passing substitution) and when the initial expression to reduce has variables. The extension of our work to cope with narrowing and strategies is left to future work. But we think that to present first a notion of rewriting with respect to which one can prove correctness and completeness of subsequent notions of narrowing and strategies is an advantage rather than a lack of our approach.

As additional future work, we plan to extend our work to the HO case as to obtain rewriting counterparts of HO-*CRWL* [8], and to relate technically *let*-rewriting with more formalisms like term graph rewriting or explicit substitutions, obtaining thus a wider picture of reduction under non-strict call-time choice.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[2] S. Antoy. Evaluation strategies for functional logic programming. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[3] S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006. To appear in ENTCS.

[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[5] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Twenty Second International Conference on Logic Programming*, pages 87–101, Seattle, WA, Aug. 2006. Springer LNCS 4079.

[6] R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.

[7] R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.

[8] J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic pro-

gramming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

[9] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.

[10] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[11] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[12] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.

[13] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[14] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[15] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Relating two semantic descriptions of functional logic programs. In *Proc. Jornadas sobre Programación y Lenguajes (PROLE'06)*, pages 31–40. CINME, 2006.

[16] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Simple Rewrite Notion for Call-time Choice Semantics. In Proc. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'07), pages 197–208. ACM, 2007.

[17] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multi-paradigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[18] F. López-Fraguas and J. Sánchez-Hernández. Functional logic programming with failure: A set-oriented view. In *Proc. International Conference on Logic for Programming and Automated Reasoning (LPAR'01)*, pages 455–469. Springer LNAI 2250, 2001.

[19] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[20] M. Marin and A. Middeldorp. New completeness results for lazy conditional narrowing. In *PPDP*, pages 120–131, 2004.

[21] D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.

[22] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Revised Lectures of the International Summer School CCL'99*, pages 202–270. Springer LNCS 2002, 2001.

[23] J. Rodríguez-Hortalá. El indeterminismo en programación lógico-funcional: un enfoque basado en reescritura. Master Thesis, DSIC, Universidad Complutense de Madrid, June 2007.

[24] J. Sánchez-Hernández. *Una aproximación al fallo constructivo en programación declarativa multiparadigma*. PhD thesis, DSIP, Universidad Complutense de Madrid, June 2004.

[25] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[26] R. d. Vado-Vírseda. A demand-driven narrowing calculus with overlapping definitional trees. In *Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'03)*, pages 213–227. ACM Press, 2003.

A. Proofs of the results

Here the reader may find full proofs of the results presented in previous sections. For more details, examples, and additional properties of *let*-rewriting we refer the reader to [23] (in Spanish).

In this appendix we use the sets $[\mathcal{P}]_\bot$ and $[\mathcal{P}]$ of partial and total c-instances of the rules of a program $\mathcal{P}$, respectively, defined as:

$$\begin{array}{lcl} [\mathcal{P}]_\bot & = & \{(f(t) \to e)\theta | \theta \in CSubst_\bot\} \\ [\mathcal{P}] & = & \{(f(t) \to e)\theta | \theta \in CSubst\} \end{array}$$

Now, the rule **(OR)** of $CRWL_{(let)}$ can be reformulated as:

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow t_1\ \ldots\ e_n \twoheadrightarrow t_n \quad e \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t}\quad (f(\bar{t}) \to e) \in [\mathcal{P}]_\bot$$

and the rule **(Fapp)** as:

$$\textbf{(Fapp)}\quad f(t_1,\ldots,t_n) \to_l e \quad \text{if } f(t_1,\ldots,t_n) \to e \in [\mathcal{P}]$$

We will use this (trivially) equivalent presentation in some of the proofs for simplicity.

When reasoning about the use of the rule **(Contx)**, we will often apply the following reasoning: if $\mathcal{C}[e] \to_l \mathcal{C}[e']$ as a consequence of $e \to_l e'$, we can always suppose that $e \to_l e'$ without applying **(Contx)**, because if it was applied then we would have $e \equiv \mathcal{C}'[e_1] \to_l \mathcal{C}'[e_2] \equiv e'$ with $e_1 \to_l e_2$, and so we can define $\mathcal{C}''[] \equiv \mathcal{C}[\mathcal{C}'[]]$ and $\mathcal{C}''[e_1] \equiv \mathcal{C}[\mathcal{C}'[e_1]] \equiv \mathcal{C}[e] \to_l \mathcal{C}[e'] \equiv \mathcal{C}[\mathcal{C}'[e_2]] \equiv \mathcal{C}''[e_2]$. Hence we can repeat that process ensuring that the rule **(Contx)** was not applied in $e \to_l e'$.

Besides, in the proofs we will often use IH to refer to the induction hypothesis.

We will use the following auxiliary results.

LEMMA 17. *For any CRWL-program P, $e \in LExp_\bot$ we have $[\![e]\!]_{CRWL_{let}} \subseteq CTerm_\bot$.*

PROOF: We have to prove that for any $e' \in LExp_\bot$ such that $\vdash_{CRWL_{let}} e \twoheadrightarrow e'$ then $e' \in CTerm_\bot$, and that is very easy to prove by a simple induction on the structure of the proof for $\vdash_{CRWL_{let}} e \twoheadrightarrow e'$. □

LEMMA 18. *For any $e, e' \in LExp_\bot$, $\mathcal{C} \in Cntxt$, $|e| \sqsubseteq |e'|$ implies $|\mathcal{C}[e]| \sqsubseteq |\mathcal{C}[e']|$.*

PROOF: A simple induction on the structure of $\mathcal{C}$, see [23] for details. □

LEMMA 19. *For any $e_1, e_2 \in LExp, X \in \mathcal{V}$, $|e_1[X/e_2]| \equiv |e_1|[X/|e_2|]$*

PROOF: A simple induction on the structure of e_1 using Lemma 5, see [23] for details. □

LEMMA 20. *For any $\sigma \in Subst_\bot$, $e, e' \in LExp_\bot$ if $e \sqsubseteq e'$ then $e\sigma \sqsubseteq e'\sigma$.*

PROOF: A simple induction on the structure of e, see [23] for details. □

A.1 For section 5.1

PROOF:[For Lemma 5] By induction of the structure of e:

- $e \equiv X$: Then
  $$(e[X/e'])\theta \equiv (X[X/e'])\theta \equiv e'\theta \equiv X[X/e'\theta]$$
  $$\equiv_{X \notin dom(\theta)} X\theta[X/e'\theta] \equiv e\theta[X/e'\theta]$$
- $e \equiv Y \in \mathcal{V} \setminus \{X\}$: Then
  $$(e[X/e'])\theta \equiv (Y[X/e'])\theta \equiv Y\theta$$
  $$\equiv_{X \notin ran(\theta)} Y\theta[X/e'\theta] \equiv e\theta[X/e'\theta]$$

- $e \equiv h(e_1, \ldots, e_n)$: Then

$$\begin{array}{l}(e[X/e'])\theta \equiv (h(e_1, \ldots, e_n))[X/e']\theta \\ \equiv h(e_1[X/e']\theta, \ldots, e_n[X/e']\theta) \\ \equiv_{IH} h(e_1\theta[X/e'\theta], \ldots, e_n\theta[X/e'\theta]) \\ \equiv (h(e_1, \ldots, e_n))\theta[X/e'\theta] \equiv e\theta[X/e'\theta]\end{array}$$

- $e \equiv let\ Y = e_1\ in\ e_2$: Then

$$\begin{array}{l}(e[X/e'])\theta \equiv (let\ Y = e_1\ in\ e_2)[X/e']\theta \\ \equiv let\ Y = e_1[X/e']\theta\ in\ e_2[X/e']\theta \\ \equiv_{IH} let\ Y = e_1\theta[X/e'\theta]\ in\ e_2\theta[X/e'\theta] \\ \equiv (let\ Y = e_1\ in\ e_2)\theta[X/e'\theta] \\ \equiv e\theta[X/e'\theta]\end{array}$$

□

PROOF:[For Lemma 6] This property is a classical result in the *CRWL* framework [9, 10]. Hence we will just extend the original proof for *CRWL*, which proceeded by induction on the structure of the proof for $\vdash_{CRWL_{let}} e \twoheadrightarrow t$, to the case where the rule applied was **(Let)**. Then we have $e \equiv let\ X = e_1\ in\ e_2$, $e \sqsubseteq e'$ implies $e' \equiv let\ X = e_1'\ in\ e_2'$ such that $e_1 \sqsubseteq e_1'$ and $e_2 \sqsubseteq e_2'$, and the proof has the shape:

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t}\ Let$$

As $e_1 \sqsubseteq e_1'$ then by IH we get $\mathcal{P} \vdash_{GORC_{let}} e_1' \rightarrow t_1$. On the other hand, as $e_2 \sqsubseteq e_2'$ then by Lemma 20 we have $e_2[X/t_1] \sqsubseteq e_2'[X/t_1]$, thus by IH we get $\mathcal{P} \vdash_{GORC_{let}} e_2'[X/t_1] \rightarrow t'$, therefore:

$$\frac{e_1' \twoheadrightarrow t_1 \quad e_2'[X/t_1] \twoheadrightarrow t'}{let\ X = e_1'\ in\ e_2' \twoheadrightarrow t'}\ Let$$

□

PROOF:[For Lemma 2]

i) This property is again a classical result in the *CRWL* framework [9, 10]. Hence we will just extend the original proof for *CRWL*, which proceeded by induction on the structure of the proof for $\vdash_{CRWL_{let}} e \twoheadrightarrow t$, to the case where the rule applied was **(Let)**. Then $e \equiv let\ X = e_1\ in\ e_2$ and the proof has the shape:

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t}\ Let$$

For $\theta \in CSubst_\perp$ then $e\theta \equiv let\ X = e_1\theta\ in\ e_2\theta$, and so

$$\frac{IH \qquad \dfrac{IH}{e_2\theta[X/t_1\theta] \equiv e_2[X/t_1]\theta \twoheadrightarrow t\theta}}{\dfrac{}{let\ X = e_1\theta\ in\ e_2\theta \twoheadrightarrow t\theta}\ Let}$$

where we have $e_2\theta[X/t_1\theta] \equiv e_2[X/t_1]\theta$ by Lemma 5, as $X \notin dom(\theta)$ y $X \notin ran(\theta)$ by the variable convention.

ii) By a simple induction on the structure of $LExp_\perp$. Every case is straightforward except for the case for $e \equiv let\ X = e_1\ in\ e_2$. Then we have $\vdash_{CRWL_{let}} e_2 \twoheadrightarrow |e_2|$ by IH, so we can apply part *i)* from Lemma 2 to get $\vdash_{CRWL_{let}} e_2[X/|e_1|] \twoheadrightarrow |e_2|[X/|e_1|]$, as $[X/|e_1|] \in CSubst_\perp$ (it is easy to check that $\forall e \in LExp_\perp.|e| \in CTerm_\perp$, again by induction on the structure of $LExp_\perp$). But then:

$$\frac{\dfrac{IH}{e_1 \twoheadrightarrow |e_1|} \quad e_2[X/|e_1|] \twoheadrightarrow |e_2|[X/|e_1|]}{let\ X = e_1\ in\ e_2 \twoheadrightarrow |e_2|[X/|e_1|] \equiv |let\ X = e_1\ in\ e_2|}\ Let$$

iii) This lemma is a trivial consequence of the following generalization of it:

LEMMA 21. *Under any program P and for any $e \in LExp_\perp$ we have that $[\![e]\!] \Subset \lambda\theta.(|e\theta|\uparrow)\downarrow$.*

The present item is an easy consequence of Lemma 21, just taking $\theta = \epsilon$. So all that is left is proving Lemma 21, for which we will use the following equivalent characterization of $(e\uparrow)\downarrow$:

$$(e\uparrow)\downarrow = \{e_1 \in LExp_\perp \mid \exists e_2 \in LExp_\perp.\, e \sqsubseteq e_2 \wedge e_1 \sqsubseteq e_2\}$$

note that $\{e_2 \in LExp_\perp \mid e \sqsubseteq e_2\}$ is precisely the set $e\uparrow$. Besides note that:

$$\begin{array}{l}[\![e]\!] \Subset \lambda\theta.(|e\theta|\uparrow)\downarrow \\ \Leftrightarrow \forall\theta \in CSubst_\perp.\, [\![e\theta]\!] \subseteq (|e\theta|\uparrow)\downarrow \\ \Leftrightarrow \forall\theta \in CSubst_\perp, t \in CTerm_\perp.\ \vdash_{CRWL_{let}} e\theta \twoheadrightarrow t \\ \qquad \Rightarrow t \in (|e\theta|\uparrow)\downarrow \\ \Leftrightarrow \forall\theta \in CSubst_\perp, t \in CTerm_\perp.\ \vdash_{CRWL_{let}} e\theta \twoheadrightarrow t \\ \qquad \Rightarrow \exists t' \in CTerm_\perp.\, |e\theta| \sqsubseteq t' \wedge t \sqsubseteq t'\end{array}$$

where $t' \in CTerm_\perp$ is implied by $|e\theta| \sqsubseteq t'$. To prove this last formulation first consider the case when $t \equiv \perp$. Then we are done with $t' \equiv |e\theta|$ because then $|e\theta| \sqsubseteq |e\theta| \equiv t'$ and $t \equiv \perp \sqsubseteq |e\theta| \equiv t'$.

For the other case we proceed by induction on the structure of e. Regarding the base cases:

- If $e \equiv \perp$ then $t \equiv \perp$ and we are in the previous case.
- If $e \equiv X \in \mathcal{V}$ then $\vdash_{CRWL_{let}} e\theta \equiv \theta(X) \twoheadrightarrow t$, and as $\theta \in CSubst_\perp$ then $\theta(X) \in CTerm_\perp$ which implies $t \sqsubseteq \theta(X)$, by a known property of *CRWL*—easy to prove by induction on the structure of $CTerm_\perp$. But then we can take $t' \equiv \theta(X)$ for which $t \sqsubseteq \theta(X) \equiv t'$ and $|e\theta| \equiv |\theta(X)| \equiv \theta(X)$, as $\theta(X) \in CTerm_\perp$ (easy to prove by induction on the structure of $CTerm_\perp$), and $\theta(X) \sqsubseteq \theta(X) \equiv t'$.
- If $e \equiv c \in DC$ then either $t \equiv \perp$ and we are in the previous case, or $t \equiv c$. But then we can take $t' \equiv c$ for which $|e\theta| \equiv c \sqsubseteq c \equiv t'$, and $t \equiv c \sqsubseteq c \equiv t'$.
- If $e \equiv f \in FS$ then $|e\theta| \equiv |f| \equiv \perp$, and so $|e\theta|\uparrow = CTerm_\perp$ and $(|e\theta|\uparrow)\downarrow = CTerm_\perp \supseteq [\![e\theta]\!]$, so we are done.

Concerning the inductive steps:

- If $e \equiv f(e_1, \ldots, e_n)$ for $f \in FS$ then $|e\theta| \equiv \perp$ and we proceed like in the case for $e \equiv f$.
- If $e \equiv c(e_1, \ldots, e_n)$ for $c \in DC$ then either $t \equiv \perp$ and we are in the previous case, or $t \equiv c(t_1, \ldots, t_n)$ such that $\forall i.\ e_i\theta \twoheadrightarrow t_i$. But then by IH we get $\forall i.\ \exists t_i'.\ |e_i\theta| \sqsubseteq t_i' \wedge t_i \sqsubseteq t_i'$, so we can take $t' \equiv c(t_1', \ldots, t_n')$ for which $|e\theta| \equiv c(|e_1\theta|, \ldots, |e_n\theta|) \sqsubseteq c(t_1', \ldots, t_n') \equiv t'$ and $t \equiv c(t_1, \ldots, t_n) \sqsubseteq c(t_1', \ldots, t_n') \equiv t'$.
- If $e \equiv let\ X = e_1\ in\ e_2$ then either $t \equiv \perp$ and we are in the previous case, or we have the following proof:

$$\frac{e_1\theta \twoheadrightarrow t_1 \quad e_2\theta[X/t_1] \twoheadrightarrow t}{e\theta \equiv let\ X = e_1\theta\ in\ e_2\theta \twoheadrightarrow t}\ Let$$

Then by IH over e_1 we get that $\exists t_1'.\ |e_1\theta| \sqsubseteq t_1' \wedge t_1 \sqsubseteq t_1'$. Hence $[X/t_1] \sqsubseteq [X/t_1']$ so by the monotonicity of $CRWL_{let}$ (see [23]) we have that $\mathcal{P} \vdash_{CRWL} e_2\theta[X/t_1] \twoheadrightarrow t$ implies $\mathcal{P} \vdash_{CRWL} e_2\theta[X/t_1'] \twoheadrightarrow t$. But then we can apply the IH over e_2 with $\theta[X/t_1']$ to get some $t' \in CTerm_\perp$

such that $t \sqsubseteq t'$ and $|e_2\theta[X/t'_1]| \sqsubseteq t'$, which implies:

$$\begin{array}{ll} t' \sqsupseteq |e_2\theta[X/t'_1]| & \\ \equiv |e_2\theta|[X/|t'_1|] & \text{by Lemma 19} \\ \equiv |e_2\theta|[X/t'_1] & \text{as } t'_1 \in CTerm_\perp(*) \\ \sqsupseteq |e_2\theta|[X/|e_1\theta|] & \text{as } |e_1\theta| \sqsubseteq t'_1 \\ \equiv |let\ X = e_1\theta\ in\ e_2\theta| \equiv |e\theta| & \end{array}$$

$(*)$ by a simple induction over the structure of $CTerm_\perp$. So we are done.

iv) By a case distinction on the rule of *let*-rewriting applied:

(Contx) As mentioned at the beginning of the Appendix, we can assume that in the step $e \to_l e'$ the rule **(Contx)** was not applied. Then by the proof of the other cases be have $|e| \sqsubseteq |e'|$, thus $|\mathcal{C}[e]| \sqsubseteq |\mathcal{C}[e']|$ by Lemma 18.

(Elim) Assume $let\ X = e_1\ in\ e_2 \to_l e_2$, con $X \notin FV(e_2)$. Then:

$$|let\ X = e_1\ in\ e_2| \equiv |e_2|[X/|e_1|] \equiv |e_2|$$

where we have $|e_2|[X/|e_1|] \equiv |e_2|$ because $X \notin FV(e_2)$, hence $X \notin FV(|e_2|)$, as no variables are introduced in the construction of the shell of an expression.

(Bind) Assume $let\ X = t\ in\ e \to_l e[X/t]$. Then

$$|let\ X = t\ in\ e| \equiv |e|[X/|t|] \equiv |e[X/t]| \quad \text{by Lemma 19}$$

(Flat) Assume $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, con $Y \notin FV(e_3)$. Then:

$$\begin{array}{ll} |let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3| & \\ \equiv |e_3|[X/(|e_2|[Y/|e_1|])] & \\ \equiv |e_3|[Y/|e_1|][X/(|e_2|[Y/|e_1|])] & \text{as } Y \notin FV(e_3) \\ \equiv |e_3|[X/|e_2|][Y/|e_1| & (*) \\ \equiv |let\ Y = e_1\ in\ let\ X = e_2\ in\ e_3| & \end{array}$$

(*) by Lemma 5 with $[e/|e_2|, \theta/[Y/|e_1|], X/X, e'/|e_2|]$, as $X \notin dom([Y/|e_1|])$ because $X \neq Y$, and $X \notin ran([Y/|e_1|])$ because than would imply $X \in FV(e_1)$ and then we would have a recursive *let* in $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3$. Therefore the variable convention ensures it is not the case.

(LetIn) Assume

$$\begin{array}{l} h(e_1, \ldots, e, \ldots, e_n) \\ \quad \to_l let\ X = e\ in\ h(e_1, \ldots, X, \ldots, e_n) \end{array}$$

for X fresh. Then we have two possibilities:

a) $h = f \in FS$: Then

$$\begin{array}{l} |let\ X = e\ in\ f(e_1, \ldots, X, \ldots, e_n)| \\ \equiv \ \perp[X/|e|] \equiv \ \perp \\ \equiv |f(e_1, \ldots, e, \ldots, e_n)| \end{array}$$

b) $h = c \in CS$: Then

$$\begin{array}{l} |let\ X = e\ in\ c(e_1, \ldots, X, \ldots, e_n)| \\ \equiv (c(|e_1|, \ldots, X, \ldots, |e_n|)[X/|e|] \\ \equiv c(|e_1|, \ldots, |e|, \ldots, |e_n|) \\ \equiv |c(e_1, \ldots, e, \ldots, e_n)| \end{array}$$

as X is fresh

(Fapp) Assume $f(t_1, \ldots, t_n) \to_l r$, then

$$|f(t_1, \ldots, t_n)| \equiv \perp \sqsubseteq |r|$$

□

PROOF:[For Lemma 3] As any ***CRWL***-proof it is also a $CRWL_{let}$-proof, then $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ obviously implies $\vdash_{CRWL_{let}} e \twoheadrightarrow t$. On the other hand if $\vdash_{CRWL_{let}} e \twoheadrightarrow t$, as $e \in Exp_\perp$—which implies it does not contain any *let*—and no rule of $CRWL_{let}$ introduces new *let*'s, then $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ is also a valid ***CRWL***-proof. □

PROOF:[For Lemma 4] We proceed by induction on the structure of the context $\mathcal{C}$.

The base case occurs when $\mathcal{C} \equiv []$, then $\mathcal{C}[e] \equiv e$ and $\mathcal{C}[e'] \equiv e'$, and so lemma holds by hypothesis.

For the inductive step we have to prove that for any $\theta \in CSubst_\perp, t \in CTerm_\perp$ if $\vdash_{CRWL_{let}} (\mathcal{C}[e])\theta \twoheadrightarrow t$ then $\vdash_{CRWL_{let}} (\mathcal{C}[e'])\theta \twoheadrightarrow t$. The case when $t \equiv \perp$ is trivial by rule **B**, regarding the other cases:

- $\mathcal{C} \equiv let\ X = \mathcal{C}'\ in\ e_1$: thus $\mathcal{C}[e] \equiv let\ X = \mathcal{C}'[e]\ in\ e_1, \mathcal{C}[e'] \equiv let\ X = \mathcal{C}'[e']\ in\ e_1$. Let $\theta \in CSubst_\perp$ be such that $(let\ X = \mathcal{C}'[e]\ in\ e_1)\theta \twoheadrightarrow t$, then it must be by the rule **Let** so we have:

$$\frac{(\mathcal{C}'[e])\theta \twoheadrightarrow t_1\ e_1\theta[X/t_1] \twoheadrightarrow t}{let\ X = (\mathcal{C}'[e])\theta\ in\ e_1\theta \twoheadrightarrow t}\ Let$$

as $[\![e]\!] \Subset [\![e']\!]$, by IH (as $\mathcal{C}'$ is part of $\mathcal{C}$) we get $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$, and so $(\mathcal{C}'[e])\theta \twoheadrightarrow t_1$ implies $(\mathcal{C}'[e'])\theta \twoheadrightarrow t_1$. But then:

$$\frac{\dfrac{IH}{(\mathcal{C}'[e'])\theta \twoheadrightarrow t_1} \quad \dfrac{hypothesis}{e_1\theta[X/t_1] \twoheadrightarrow t}}{let\ X = (\mathcal{C}'[e'])\theta\ in\ e_1\theta \twoheadrightarrow t}\ Let$$

- $\mathcal{C} \equiv let\ X = e_1\ in\ \mathcal{C}'$: thus $\mathcal{C}[e] \equiv let\ X = e_1\ in\ \mathcal{C}'[e]$. Let $\theta \in CSubst_\perp$ be such that $(let\ X = e_1\ in\ \mathcal{C}'[e])\theta \twoheadrightarrow t$, then it must be by the rule **Let** so we have:

$$\frac{e_1\theta \twoheadrightarrow t_1\ (\mathcal{C}'[e])\theta[X/t_1] \twoheadrightarrow t}{let\ X = e_1\theta\ in\ (\mathcal{C}'[e])\theta \twoheadrightarrow t}\ Let$$

as $[\![e]\!] \Subset [\![e']\!]$ by IH $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$, which combined with $([X/t_1] \circ \theta) \in CSubst_\perp$ implies $(\mathcal{C}'[e'])\theta[X/t_1] \twoheadrightarrow t$. But then:

$$\frac{\dfrac{hypothesis}{e_1\theta \twoheadrightarrow t_1} \quad \dfrac{IH}{(\mathcal{C}'[e'])\theta[X/t_1] \twoheadrightarrow t}}{let\ X = e_1\theta\ in\ (\mathcal{C}'[e'])\theta \twoheadrightarrow t}\ Let$$

- $\mathcal{C} \equiv h(\ldots, \mathcal{C}', \ldots)$: thus $\mathcal{C}[e] \equiv h(\ldots, \mathcal{C}'[e], \ldots)$. Let $\theta \in CSubst_\perp$ be such that $h(\ldots, \mathcal{C}'[e], \ldots)\theta \twoheadrightarrow t$. Then we have two possibilities:

a) $h \equiv f \in FS$: then the rule applied must be **OR** so we have:

$$\frac{e_1\theta \twoheadrightarrow t_1\ \ldots\ (\mathcal{C}'[e])\theta \twoheadrightarrow t'\ \ldots\ e_n\theta \twoheadrightarrow t_n\ r \twoheadrightarrow t}{f(e_1\theta, \ldots, (\mathcal{C}'[e])\theta, \ldots, e_n\theta) \twoheadrightarrow t}\ OR$$

where $(f(t_1, \ldots, t', \ldots, t_n) = r) \in [\mathcal{P}]_\perp$. As $[\![e]\!] \Subset [\![e']\!]$ by IH we get $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$, thus $(\mathcal{C}'[e'])\theta \twoheadrightarrow t'$ and so:

$$\frac{\dfrac{hypothesis}{e_1\theta \twoheadrightarrow t_1} \ \ldots\ \dfrac{IH}{(\mathcal{C}'[e'])\theta \twoheadrightarrow t'}\ \ldots\ \dfrac{hypothesis}{e_n\theta \twoheadrightarrow t_n} \quad \dfrac{hypothesis}{r \twoheadrightarrow t}}{f(e_1\theta, \ldots, (\mathcal{C}'[e'])\theta, \ldots, e_n\theta) \twoheadrightarrow t}\ OR$$

b) $h \equiv c \in CS$: then the rule applied must be **DC** so we have:

$$\frac{e_1\theta \twoheadrightarrow t_1\ \ldots\ (\mathcal{C}'[e])\theta \twoheadrightarrow t'\ \ldots\ e_n\theta \twoheadrightarrow t_n}{c(e_1\theta, \ldots, (\mathcal{C}'[e])\theta, \ldots, e_n\theta) \twoheadrightarrow c(t_1, \ldots, t', \ldots, t_n)}\ DC$$

As $[\![e]\!] \Subset [\![e']\!]$ we can apply the IH to get $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$, thus $(\mathcal{C}'[e'])\theta \twoheadrightarrow t'$ and so:

$$\frac{\frac{}{e_1\theta \twoheadrightarrow t_1}\ hypothesis \quad \ldots \quad \frac{}{(\mathcal{C}'[e'])\theta \twoheadrightarrow t'}\ IH \quad \ldots \quad \frac{}{e_n\theta \twoheadrightarrow t_n}\ hypothesis}{c(e_1\theta,\ldots,(\mathcal{C}'[e'])\theta,\ldots,e_n\theta) \twoheadrightarrow c(t_1,\ldots,t',\ldots,t_n)}\ DC$$

□

PROOF:[For Theorem 3] We assume $\theta \in CSubst_\perp$ such that $e'\theta \twoheadrightarrow t$. We must prove that $e\theta \twoheadrightarrow t$. The case where $e'\theta \twoheadrightarrow \perp$ holds trivially using the rule **B**, so we will prove the other by a case distinction on the rule of the *let* calculus applied:

(Contx) As mentioned at the beginning of the Appendix, we can assume that in the step $e \to_l e'$ the rule **(Contx)** was not applied. Then, by the proof of the other cases, $[\![e']\!] \Subset [\![e]\!]$, and by Lemma 4, $[\![\mathcal{C}[e']]\!] \Subset [\![\mathcal{C}[e]]\!]$, and we are done.

(Elim) Assume $let\ X = e_1\ in\ e_2 \to_l e_2$ and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} e_2\theta \twoheadrightarrow t$:

$$\frac{\frac{}{e_1\theta \twoheadrightarrow \perp}\ B \quad \frac{}{e_2\theta[X/\perp] \equiv e_2\theta \twoheadrightarrow t}\ Hypothesis}{let\ X = e_1\theta\ in\ e_2\theta \twoheadrightarrow t}\ Let$$

We know that $X \notin ran(\theta)$ because of the way we have defined substitutions [5]. Then as $X \notin FV(e_2)$ for the condition of **(Elim)**, $X \notin FV(e_2\theta)$ and so $e_2\theta[X/\perp] \equiv e_2\theta$.

(Bind) Assume $let\ X = t_1\ in\ e \to_l e[X/t_1]$ and $\theta \in CSubst_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} (e[X/t_1])\theta \twoheadrightarrow t$:

$$\frac{\frac{}{t_1\theta \twoheadrightarrow t_1\theta}\ DC* \quad \frac{}{e\theta[X/t_1\theta] \equiv (e[X/t_1])\theta \twoheadrightarrow t}\ Hyp}{let\ X = t_1\theta\ in\ e\theta \twoheadrightarrow t}\ Let$$

By our definition of substitutions we assume $X \notin dom(\theta)$ and $X \notin ran(\theta)$, so by Lemma 5 we have $e\theta[X/t_1\theta] \equiv (e[X/t_1])\theta$. Besides, "rule" $[DC*]$ refers to the fact that $\forall t \in CTerm_\perp . \mathcal{P} \vdash_{CRWL_{let}} t \twoheadrightarrow t$ (very easy to prove).

(Flat) Assume $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} (let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3))\theta \twoheadrightarrow t$. This proof must be must be of the shape of:

$$\frac{\frac{}{e_1\theta \twoheadrightarrow t_1} \quad \frac{\frac{e_2\theta[Y/t_1] \twoheadrightarrow t_2 \quad (e_3\theta[Y/t_1] \equiv e_3\theta)[X/t_2] \twoheadrightarrow t}{(let\ X = e_2\theta\ in\ e_3\theta)[Y/t_1] \twoheadrightarrow t}\ Let}{let\ Y = e_1\theta\ in\ (let\ X = e_2\theta\ in\ e_3\theta) \twoheadrightarrow t}\ Let$$

for some $t_1, t_2 \in CTerm_\perp$. Besides, because of the way we have defined substitutions, $Y \notin ran(\theta)$, so as by the condition of **(Flat)**, $Y \notin FV(e_3)$, then $Y \notin FV(e_3\theta)$ and we can say $e_3\theta[Y/t_1] \equiv e_3\theta$. So:

$$\frac{\frac{\frac{}{e_1\theta \twoheadrightarrow t_1}\ Hyp \quad \frac{}{e_2\theta[Y/t_1] \twoheadrightarrow t_2}\ Hyp}{let\ Y = e_1\theta\ in\ e_2\theta \twoheadrightarrow t_2}\ Let \quad \frac{}{e_3\theta[X/t_2] \twoheadrightarrow t}\ Hyp}{let\ X = (let\ Y = e_1\theta\ in\ e_2\theta)\ in\ e_3\theta \twoheadrightarrow t}\ Let$$

(LetIn) Assume $h(d_1,\ldots,e,\ldots,d_n) \to_l let\ X = e\ in\ h(d_1,\ldots,X,\ldots,d_n)$ and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} (let\ X = e\ in\ h(d_1,\ldots,X,\ldots,d_n))\theta \twoheadrightarrow t$. This proof will reduce to the proofs $e\theta \twoheadrightarrow t_1$ and $h(d_1,\ldots,X,\ldots,d_n)\theta[X/t_1] \twoheadrightarrow t$ for some $t_1 \in CTerm_\perp$. By the variable convention $X \notin dom(\theta)$ and $X \notin ran(\theta)$, so as X is fresh then $\forall i.X \notin FV(d_i\theta)$, hence $h(d_1,\ldots,X,\ldots,d_n)\theta[X/t_1] \equiv h(d_1\theta,\ldots,t_1,\ldots,d_n\theta)$. Now there are two possible cases:

a) $h = c \in DC$, then $h(d_1\theta,\ldots,t_1,\ldots,d_n\theta) \twoheadrightarrow t$ must proved by (DC) as:

$$\frac{d_1\theta \twoheadrightarrow s_1\ \ldots\ t_1 \twoheadrightarrow t'_1\ \ldots\ d_n\theta \twoheadrightarrow s_n}{c(d_1\theta,\ldots,t_1,\ldots,d_n\theta) \twoheadrightarrow c(s_1,\ldots,t'_1,\ldots,s_n) \equiv t}$$

for some $s_1,\ldots,s_n,t'_1 \in CTerm_\perp$. As $\forall t \in CTerm_\perp$, $t \twoheadrightarrow t'$ implies $t' \sqsubseteq t$ (easy to prove, as only **B**, **RR** and **DC** could be applied), then $t'_1 \sqsubseteq t_1$, and so as $e\theta \twoheadrightarrow t_1$, by Lemma 6 we have $e\theta \twoheadrightarrow t'_1$. Then we have proofs for $d_1\theta \twoheadrightarrow s_1 \ldots e\theta \twoheadrightarrow t'_1 \ldots d_n\theta \twoheadrightarrow s_n$, and with **DC** we can build a proof for $c(d_1\theta,\ldots,e\theta,\ldots,d_n\theta) \twoheadrightarrow c(s_1,\ldots,t'_1,\ldots,s_n) \equiv t$.

b) $h = f \in FS$, then $h(d_1\theta,\ldots,t_1,\ldots,d_n\theta) \twoheadrightarrow t$ must be:

$$\frac{d_1\theta \twoheadrightarrow s_1\ \ldots\ t_1 \twoheadrightarrow t'_1\ \ldots\ d_n\theta \twoheadrightarrow s_n \quad r \twoheadrightarrow t}{f(d_1\theta,\ldots,t_1,\ldots,d_n\theta) \twoheadrightarrow t}\ OR$$

for some $(f(s_1,\ldots,t'_1,\ldots,s_n) \to r) \in [\mathcal{P}]_\perp$. Again as $\forall t \in CTerm_\perp, t \twoheadrightarrow t'$ implies $t' \sqsubseteq t$, then $t'_1 \sqsubseteq t_1$, and so as $e\theta \twoheadrightarrow t_1$, by Lemma 6 we have $e\theta \twoheadrightarrow t'_1$. So we have proofs for $d_1\theta \twoheadrightarrow s_1 \ldots e\theta \twoheadrightarrow t'_1 \ldots d_n\theta \twoheadrightarrow s_n$ and $r \twoheadrightarrow t$ and then with **OR** we can build the proof for $f(d_1\theta,\ldots,e\theta,\ldots,d_n\theta) \twoheadrightarrow t$.

(Fapp) Assume $f(t_1,\ldots,t_n) \to_l r$ with $(f(p_1,\ldots,p_n) = e)\sigma \in [\mathcal{P}]$ such that $\forall i.p_i\sigma = t_i$ and $e\sigma = r$, and $\theta \in CSubts_\perp$ such that $\mathcal{P} \vdash_{CRWL_{let}} r\theta \twoheadrightarrow t$. Then as $\theta \circ \sigma \in CSubts_\perp, \forall i.p_i\sigma\theta = t_i\theta$ and $e\sigma\theta = r\theta$ we conclude $(f(p_1,\ldots,p_n) = e)\sigma\theta \in [\mathcal{P}]_\perp$ and so:

$$\frac{\frac{}{t_1\theta \twoheadrightarrow t_1\theta}\ DC^* \quad \ldots \quad \frac{}{t_n\theta \twoheadrightarrow t_n\theta}\ DC^* \quad \frac{}{r\theta \twoheadrightarrow t}\ hypothesis}{f(t_1\theta,\ldots,t_n\theta) \twoheadrightarrow t}\ OR$$

□

A.2 For section 5.2

The next result is a well known result in the scope of *CRWL* and will be used to prove Lemma 7.

LEMMA 22. *Let linear $p \in CTerm$, and $t_1 \in CTerm_\perp$, $t_2 \in CTerm$, $\theta \in CSubst_\perp$. Then $p\theta = t_1$ and $t_1 \sqsubseteq t_2$ implies $\exists\theta' \in CSubst$ such that $p\theta' = t_2$ and $\theta \sqsubseteq \theta'$.*

We will also use the following results.

LEMMA 23. *For any $\sigma \in Subst_\perp, e \in LExp_\perp$ if $|e| \equiv \perp$ then $|e\sigma| \equiv \perp$.*

PROOF: A simple induction on the structure of e. □

LEMMA 24. *For any $\theta \in CSubst_\perp, t \in CTerm_\perp$ we have that $t\theta \in CTerm_\perp$.*

PROOF: A simple induction on the structure of t. □

PROOF:[For Lemma 7] By induction on the structure of e:

Base Case :

- $e \equiv Y \in \mathcal{V}$: then $Y \to_l^0 Y$, ok with $\overline{X} \equiv \emptyset$.

[5] Actually, to prove this theorem properly, we cannot restrict the substitution to fulfil these restrictions, so in fact we rename the bound variables in an α-conversion fashion and use the equivalence $e[X/t] \equiv e[X/Y][Y/t]$ (with Y the new bound variable), to use the hypothesis. We will assume this convention from now on.

- $e \equiv h \in \Sigma$: then $h \to_l^0 h$, ok with $\overline{X} \equiv \emptyset$.

Inductive Step :

- $e \equiv h(e_1, \ldots, e_n)$: Let us do it for just one argument, for $h(e_1)$. If $e_1 \in CTerm$ then we are done with $\overline{X} \equiv \emptyset$ and $h(e_1) \to_l^0 h(e_1)$, note how the arguments which are c-terms remain in the same position.
  On the other hand let us suppose that $e_1 \notin CTerm$. Then by IH $e_1 \to_l^* let\ \overline{X_1 = a_1}\ in\ b_1$ and we have two possibilities. If $\overline{X_1} \equiv \emptyset$ then we did $e \equiv h(e_1) \to_l^* h(b_1)$, and there are several possible cases:

  a) $b_1 \equiv f_1(\overline{t_1})$ for $f_1 \in FS$: then

  $$h(b_1) \equiv h(f_1(\overline{t_1})) \to_l let\ Y_1 = f_1(\overline{t_1})\ in\ h(Y_1)$$

  by **(LetIn)**, and we are done as $f_1 \in FS$ implies $|f_1(\overline{t_1})| = \bot$.

  b) $b_1 \in \mathcal{V}$ or $b_1 \equiv c_1(\overline{t_1})$ for $c_1 \in DC$: then $b_1 \in CTerm$ because if $b_1 \in \mathcal{V}$ we can apply $\mathcal{V} \subseteq CTerm$, otherwise we have $\overline{t_1} \subseteq CTerm$ by IH, which combined with $c_1 \in DC$ implies $c_1(\overline{t_1}) \in CTerm$. Thus we are done with $\overline{X} \equiv \emptyset$.

  Conversely if $\overline{X_1} \not\equiv \emptyset$ then $h(e_1) \to_l^* h(let\ \overline{X_1 = a_1}\ in\ b_1)$

  $$\begin{array}{l} h(e_1) \to_l^* h(let\ \overline{X_1 = a_1}\ in\ b_1) \\ \to_l^* let\ Y_1 = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ h(Y_1) \\ \to_l^* let\ \overline{X_1 = a_1}\ in\ let\ Y_1 = b_1\ in\ h(Y_1) \end{array}$$

  by IH, **(LetIn)** and several applications of **(Flat)**.

  Then there are two possible cases:

  a) $b_1 \equiv f_1(\overline{t_1})$ for $f_1 \in FS$: then we are done as $\forall a_i \in \overline{a}.|a_i| = \bot$ by the IH, and $|f_1(\overline{t_1})| = \bot$.

  b) $b_1 \in \mathcal{V}$ or $b_1 \equiv c_1(\overline{t_1})$ for $c_1 \in DC$: then $b_1 \in CTerm$ and so we can apply (Bind) as follows

  $$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ Y_1 = b_1\ in\ h(Y_1) \\ \to_l let\ \overline{X_1 = a_1}\ in\ h(b_1) \end{array}$$

  and we are done as $\forall a_i \in \overline{a}.|a_i| = \bot$ by the IH.

  Using this techniques we can extend the proof to the case when h has more than one argument. The first thing we had to do then is applying the IH to each argument, then proceeding with the case distinction for each argument. Note that the **(Bind)** step used for the case $\overline{X_1} \not\equiv \emptyset$, *b)* only affects the fresh variable created for tha corresponding argument in the previous **(LetIn)** step, so different arguments do not interfere with each other.

- $e = let\ X = e_1\ in\ e_2$: first of all we apply the IH to both e_1 and e_2, so we can do:

  $$e \to_l^* let\ X = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ let\ \overline{X_2 = a_2}\ in\ b_2$$

  If $\overline{X_1} = \emptyset$ we have several possible cases:

  a) $b_1 \equiv f_1(\overline{t_1})$ for $f_1 \in FS$: then we have:

  $$e \to_l^* let\ X = f_1(\overline{t_1})\ in\ let\ \overline{X_2 = a_2}\ in\ b_2$$

  and we are done as $|\overline{a_2}| = \overline{\bot}$ by IH, $|f_1(\overline{t_1})| = \bot$ as $f_1 \in FS$, and b_2 fulfils the conditions of the Lemma by IH.

  b) $b_1 \in \mathcal{V}$ or $b_1 \equiv c_1(\overline{t_1})$ for $c_1 \in DC$: then $b_1 \in CTerm$ and so we can apply (Bind) to get:

  $$\begin{array}{l} e \to_l^* let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \\ \to_l let\ \overline{X_2 = a_2[X/b_1]}\ in\ b_2[X/b_1] \end{array}$$

  and we are done as $|\overline{a_2}| = \overline{\bot}$ by IH combined with Lemma 23 implies $|\overline{a_2[X/b_1]}| = \overline{\bot}$; and $b_2[X/b_1]$ fulfils the conditions of the Lemma, because by IH we have the following possibilities:

  i) $b_2 \equiv X$: then $b_2[X/b_1] \equiv b_1$, which fulfils the conditions of the Lemma by IH.

  ii) $b_2 \in (\mathcal{V} \setminus \{X\})$: then $b_2[X/b_1] \equiv b_2 \in \mathcal{V}$.

  iii) $b_2 \equiv h_2(\overline{t_2})$ for $\overline{t_2} \subseteq CTerm$: then as $b_1 \in CTerm$, by Lemma 24 we obtain $b_2[X/b_1] \equiv h_2(\overline{t_2[X/b_1]})$ for $\overline{t_2[X/b_1]} \subseteq CTerm$.

  Conversely if $\overline{X_1} \not\equiv \emptyset$ then

  $$\begin{array}{l} e \to_l^* let\ X = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \\ \to_l^* let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \end{array}$$

  by several applications of **(Flat)**.

  Then there are several possible cases:

  a) $b_1 \equiv f_1(\overline{t_1})$ for $f_1 \in FS$: then we have:

  $$\begin{array}{c} e \to_l^* let\ \overline{X_1 = a_1}\ in\ let\ X = f_1(\overline{t_1}) \\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \end{array}$$

  and we are done as $|\overline{a_1}| = \overline{\bot}$ by IH, $|f_1(\overline{t_1})| = \bot$ as $f_1 \in FS$, $|\overline{a_2}| = \overline{\bot}$ by IH, and b_2 fulfils the conditions of the Lemma by IH.

  b) $b_1 \in \mathcal{V}$ or $b_1 \equiv c_1(\overline{t_1})$ for $c_1 \in DC$: then $b_1 \in CTerm$ and so we can apply **(Bind)** to get:

  $$\begin{array}{l} e \to_l^* let\ \overline{X_1 = a_1}\ in\ let\ X = b_1 \\ \quad in\ let\ \overline{X_2 = a_2}\ in\ b_2 \\ \to_l let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_2 = a_2[X/b_1]}\ in\ b_2[X/b_1] \end{array}$$

  and we are done as $|\overline{a_1}| = \overline{\bot}$ by IH, $|\overline{a_2}| = \overline{\bot}$ by IH combined with Lemma 23 implies $|\overline{a_2[X/b_1]}| = \overline{\bot}$; and $b_2[X/b_1]$ fulfils the conditions of the Lemma by the same case distinction used for $\overline{X_1} \not\equiv \emptyset$ part *b)*.

□

PROOF:[For Lemma 8] By induction on the size s of the *CRWL*-proof, that we measure as the number of *CRWL* rules applied:

Base Case: $s = 1$. Let us see which rule was applied:

B This contradicts the hypothesis because then $t \equiv \bot$, so we are done. In the rest of the proof we will assume that $t \not\equiv \bot$ because otherwise we would be in this case.

RR Then we have $\mathcal{P} \vdash_{CRWL} X \twoheadrightarrow X$. But then $X \to_l^0 X$ and $X \sqsubseteq X \equiv |X|$, so we are done with $\overline{X} = \emptyset$.

DC Then we have $\mathcal{P} \vdash_{CRWL} c \twoheadrightarrow c$. But then $c \to_l^0 c$ and $c \sqsubseteq c \equiv |c|$, so we are done with $\overline{X} = \emptyset$.

Inductive Step: $s > 1$. Let us see which rule was applied:

DC Then we have $e \equiv c(e_1, \ldots, e_n)$ and the *CRWL*-proof has the form:

$$\frac{e_1 \twoheadrightarrow t_1, \ldots, e_n \twoheadrightarrow t_n}{c(e_1, \ldots, e_n) \twoheadrightarrow c(t_1, \ldots, t_n)}\ DC$$

In the general case we can have $t_i = \bot$ for some i's and $t_j \neq \bot$ for the remaining ones. For simplicity we consider the case the

case $n = 2$ with $t_1 = \bot$ and $t_2 \neq \bot$ (it is easy to extend the result for the general case), we have $\mathcal{P} \vdash_{CRWL} c(e_1, e_2) \twoheadrightarrow c(\bot, t_2)$. Then by IH over the second argument we have $e_2 \to_l^* let\ \overline{X_2 = a_2}\ in\ t_2'$, with $t_2' \in CTerm$, $|a_{2_i}| = \bot$ for every a_{2_i} and $|let\ \overline{X_2 = a_2}\ in\ t_2'| = t_2'[\overline{X_2/\ \bot}] \sqsupseteq t_2$. So:

$c(e_1, e_2) \to_l^* c(e_1, let\ \overline{X_2 = a_2}\ in\ t_2')$ by IH
$\to_l^* let\ Y = (let\ \overline{X_2 = a_2}\ in\ t_2')\ in\ c(e_1, Y)$ by (LetIn)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ Y = t_2'\ in\ c(e_1, Y))$ by (Flat)*
$\to_l^* let\ \overline{X_2 = a_2}\ in\ c(e_1, t_2')$ by (Bind)

Then there are several possible cases:

a) $e_1 = f_1(\overline{e_1})$: Then $let\ \overline{X_2 = a_2}\ in\ c(f_1(\overline{e_1}), t_2') \to_l let\ \overline{X_2 = a_2}\ in\ let\ Z = f_1(\overline{e_1})\ in\ c(Z, t_2')$, by **(LetIn)**. So we are done as $|a_{2_i}| = \bot$ for every a_{2_i} by the IH, $|f_1(\overline{e_1})| = \bot$ and

$$|let\ \overline{X_2 = a_2}\ in\ let\ Z = f_1(\overline{e_1})\ in\ c(Z, t_2')| = c(Z, t_2')[\overline{X_2/\ \bot}, Z/\ \bot] \sqsupseteq c(\bot, t_2)$$

because $t_2'[\overline{X_2/\ \bot}] \sqsupseteq t_2$ by the IH, and Z is fresh and so does not appear in t_2'.

b) $e_1 = t_1' \in CTerm$; Then we are done as $|a_{2_i}| = \bot$ for every a_{2_i} by the IH, and

$$|let\ \overline{X_2 = a_2}\ in\ c(t_1', t_2')| = c(t_1', t_2')[\overline{X_2/\ \bot}] \sqsupseteq c(\bot, t_2)$$

because $t_2'[\overline{X_2/\ \bot}] \sqsupseteq t_2$ by the IH.

c) $e_1 = c_1(\overline{e_1}) \notin CTerm$: Then by Lemma 7, $c_1(\overline{e_1}) \to_l^* let\ \overline{X_1 = a_1}\ in\ c_1(\overline{t_1})$ such that $|a_{1_i}| = \bot$ for every a_{1_i}. But then:

$let\ \overline{X_2 = a_2}\ in\ c(c_1(\overline{e_1}), t_2')$
$\to_l^* let\ \overline{X_2 = a_2}\ in\ c(let\ \overline{X_1 = a_1}\ in\ c_1(\overline{t_1}), t_2')$ (by Lemma 7)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ Y = (let\ \overline{X_1 = a_1}\ in\ c_1(\overline{t_1}))\ in\ c(Y, t_2')$ (by LetIn)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ let\ Y = c_1(\overline{t_1})\ in\ c(Y, t_2')$ (by Flat*)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ c(c_1(\overline{t_1}), t_2')$ by (Bind), as Y is fresh.

Then we are done as $|a_{1_i}| = \bot$ for every a_{1_i} by Lemma 7, $|a_{2_i}| = \bot$ for every a_{2_i} by the IH, and

$$|let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ c(c_1(\overline{t_1}), t_2')| = c(c_1(\overline{t_1}), t_2')[\overline{X_1/\ \bot}][\overline{X_2/\ \bot}] \sqsupseteq c(\bot, t_2)$$

because $t_2'[\overline{X_2/\ \bot}] \sqsupseteq t_2$ by the IH, and $\overline{X_1}$ are fresh and so do not appear in t_2'.

d) $e_1 = let\ X = e_{11}\ in\ e_{12}$: this case is impossible as in Lemma 8 we assume $e \in Term$, without lets!

OR If f has no arguments ($n = 0$) then we have:

$$\frac{r\theta \twoheadrightarrow t}{f \twoheadrightarrow t}\ OR$$

with $(f \twoheadrightarrow r\theta) \in [\mathcal{P}]_\bot$. Let us define $\theta' \in CSubst$ as the substitution which is equal to θ except that every $\bot$ introduced by θ is replaced with some constructor symbol or variable. Then $\theta \sqsubseteq \theta'$, so by Lemma 6 we have $\mathcal{P} \vdash_{CRWL} r\theta' \twoheadrightarrow t$ with a proof of the same size. But then appliying the IH to this proof we get $r\theta' \to_l^* let\ \overline{X = a}\ in\ t'$ under the conditions of the lemma. But then $f \to_l e\theta' \to_l^* let\ \overline{X = a}\ in\ t'$ applying **(Fapp)** in the first step, so we are done.

If $n > 0$, we will proceed as in the case for **(DC)**, doing a preliminary version for $\mathcal{P} \vdash_{CRWL} f(e_1, e_2) \twoheadrightarrow t$ which can be easily extended for the general case. Then we have:

$$\frac{e_1 \twoheadrightarrow \bot\ \ e_2 \twoheadrightarrow t_2\ r \twoheadrightarrow t}{f(e_1, e_2) \twoheadrightarrow t}\ OR$$

such that $t_2 \neq \bot$, and with $(f(p_1, p_2) = e)\theta \in [\mathcal{P}]_\bot$ such that $p_1\theta = \bot$, $p_2\theta = t_2$ and $e\theta = r$. Then applying the IH to $\mathcal{P} \vdash_{CRWL} e_2 \twoheadrightarrow t_2$ we get that $e_2 \to_l^* let\ \overline{X_2 = a_2}\ in\ t_2'$ such that $|a_{2_i}| = \bot$ for every a_{2_i} and $|let\ \overline{X_2 = a_2}\ in\ t_2'| = t_2'[\overline{X_2/\ \bot}] \sqsupseteq t_2$. So:

$f(e_1, e_2) \to_l^* f(e_1, let\ \overline{X_2 = a_2}\ in\ t_2')$ by the IH
$\to_l^* let\ Y = (let\ \overline{X_2 = a_2}\ in\ t_2')\ in\ f(e_1, Y)$ by (LetIn)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ Y = t_2'\ in\ f(e_1, Y)$ by (Flat)*
$\to_l^* let\ \overline{X_2 = a_2}\ in\ f(e_1, t_2')$ by (Bind)

Then applying Lemma 7 we get

$$f(e_1, t_2') \to_l^* let\ \overline{X_1 = a_1}\ in\ f(t_1', t_2')$$

such that $|a_{1_i}| = \bot$ for every a_{1_i}. Now as $t_2'[\overline{X_2/\ \bot}] \sqsupseteq t_2$ then $(t_1', t_2') \sqsupseteq (\bot, t_2)$, so by Lemma 22 there must exists $\theta' \in CSubst$ such that $\theta \sqsubseteq \theta'$ and $(p_1, p_2)\theta' = (t_1', t_2')$. Then by Lemma 6, as $\mathcal{P} \vdash_{CRWL} r \equiv e\theta \twoheadrightarrow t$ then $\mathcal{P} \vdash_{CRWL} e\theta' \twoheadrightarrow t$ with a proof of the same size. As $\theta' \in CSubst$ and $e \in Term$ (because it is part of the program) then $e\theta' \in Term$ and we can apply the IH to that Crwl-proof getting that $e\theta' \to_l^* let\ \overline{X = a}\ in\ t'$ such that $|a_i| = \bot$ for every a_i and $|let\ \overline{X = a}\ in\ t'| = t'[\overline{X/\ \bot}] \sqsupseteq t$. So:

$let\ \overline{X_2 = a_2}\ in\ f(e_1, t_2')$
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ f(t_1', t_2')$ (by Lemma 7)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ e\theta'$ (by Fapp)
$\to_l^* let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ let\ \overline{X = a}\ in\ t'$ by 2^{nd} IH.

Then $|a_{2_i}| = \bot$ for every a_{2_i} by IH, $|a_{1_i}| = \bot$ for every a_{1_i} by Lemma 7 and $|a_i| = \bot$ for every a_i by IH. As the variables in $\overline{X_1} \cup \overline{X_2}$ are fresh variables introducted by the let-calculus, none of those can appear in t. So $t'[\overline{X/\ \bot}] \sqsupseteq t$ implies that $\forall p \in O(t')$ such that $t'|_p = Y$ such that $Y \in \overline{X_1} \cup \overline{X_2}$ then $t|_p = \bot$. So $|let\ \overline{X_2 = a_2}\ in\ let\ \overline{X_1 = a_1}\ in\ let\ \overline{X = a}\ in\ t'| = t'[\overline{X/\ \bot}][\overline{X_1/\ \bot}][\overline{X_2/\ \bot}] \sqsupseteq t$. □

A.3 For section 6.1

PROOF:[For Lemma 12] We proceed by a case distinction over the rule of *let*-rewriting applied.

(Contx) Then we have $e \equiv \mathcal{C}[a] \to_l \mathcal{C}[a'] \equiv e'$ for $a \to_l a'$. As mentioned at the beginning of the Appendix, we can assume that in the step $a \to_l a'$ the rule **(Contx)** was not applied. Then by the proof of the other cases we get $\widehat{a} \to^* \widehat{a'}$. But it can be easily proved that for any context $\mathcal{C}$, $e, e' \in LExp$ we have that $\widehat{e} \to^* \widehat{e'}$ implies $\widehat{\mathcal{C}[e]} \to^* \widehat{\mathcal{C}[e']}$, by a simple induction on the structure of context and using Lemma 10, closedness under substitutions of term rewriting and compatibility with Σ-operations of term rewriting (see [23] for details). We can apply this result to $\widehat{a} \to^* \widehat{a'}$ thus getting $\widehat{e} \equiv \widehat{\mathcal{C}[a]} \to^* \widehat{\mathcal{C}[a']} \equiv \widehat{e'}$.

(LetIn) Then we have:

$e \equiv h(e_1, \ldots, s, \ldots, e_n)$
$\to_l let\ X = s\ in\ h(e_1, \ldots, X, \ldots, e_n) \equiv e'$

for X fresh. But then $\widehat{e'} \equiv h(\widehat{e_1},\ldots,X,\ldots,\widehat{e_n})[X/\widehat{s}] \equiv h(\widehat{e_1},\ldots,\widehat{s},\ldots,\widehat{e_n}) \equiv \widehat{e'}$, as$X$ is fresh, thus $\widehat{e} \to^0 \widehat{e} \equiv \widehat{e'}$.

(Flat) Then we have:

$$e \equiv let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \\ \to_l let\ Y = e_1\ in\ let\ X = e_2\ in\ e_3 \equiv e'$$

with $Y \notin FV(e_3)$. As the variable convention forbids recursive *let*'s, we also have $X \notin FV(e_1)$, and so by Lemma 9 we get $X \notin var(\widehat{e_1})$ and $Y \notin var(\widehat{e_3})$. But then:

$$\begin{array}{ll} \widehat{e} \equiv \widehat{e_3}[X/(\widehat{e_2}[Y/\widehat{e_1}])] & \\ \equiv \widehat{e_3}[Y/\widehat{e_1}][X/(\widehat{e_2}[Y/\widehat{e_1}])] & \text{as } Y \notin var(\widehat{e_3}) \\ \equiv \widehat{e_3}[X/\widehat{e_2}][Y/\widehat{e_1}] & \text{by Lemma 5} \\ \equiv \widehat{e'} & \end{array}$$

Thus $\widehat{e} \to^0 \widehat{e} \equiv \widehat{e'}$.

(Bind) Then we have:

$$e \equiv let\ X = t\ in\ e_1 \to_l e_1[X/t] \equiv e'$$

for $t \in CTerm$. But then $\widehat{e} \equiv \widehat{e_1}[X/\widehat{t}] \equiv \widehat{e_1[X/t]} \equiv \widehat{e'}$, by Lemma 11, thus $\widehat{e} \to^0 \widehat{e} \equiv \widehat{e'}$.

(Elim) Then we have:

$$e \equiv let\ X = e_1\ in\ e_2 \to_l e_2 \equiv e'$$

with $X \notin FV(e_2)$. But then by Lemma 9 we get $var(\widehat{e_2}) \subseteq FV(e_2)$, hence $X \notin var(\widehat{e_2})$, and so $\widehat{e} \equiv \widehat{e_2}[X_p/\widehat{e_1}] \equiv \widehat{e_2} \equiv \widehat{e'}$, thus $\widehat{e} \to^0 \widehat{e} \equiv \widehat{e'}$.

(Fapp) Then we have:

$$e \equiv f(\bar{t}) \to_l s \equiv e' \text{ for } (f(\bar{t}) \to s) \in [\mathcal{P}]$$

As $(f(\bar{t}) \to s) \in [\mathcal{P}]$ then there must exists a fresh variant $(f(\overline{p}) \to r) \in \mathcal{P}$ and a substitution $\sigma \in CSubst$ such that $(f(\overline{p}) \to r)\sigma \equiv (f(\bar{t}) \to s)$. Besides, as $(f(\bar{t}) \to s) \in [\mathcal{P}]$ then $f(\bar{t}), s \in Exp$, but then by Lemma 9 we get $f(\bar{t}) \equiv \widehat{f(\bar{t})}$ and $s \equiv \widehat{s}$. And now $\widehat{e} \equiv \widehat{f(\bar{t})} \equiv f(\bar{t}) \equiv (f(\overline{p}))\sigma \to r\sigma \equiv s \equiv \widehat{s} \equiv \widehat{e'}$, by a term rewriting step.

□

A.4 For section 6.2

PROOF:[For Lemma 13] Assume a deterministic program and $e \to_l^* e_1$ y $e \to_l^* e_2$. Then by Theorem 4 we have $\forall i \in \{1,2\}.|e_i| \in [\![e]\!]$. But then as the program is deterministic then $\exists t_3 \in [\![e]\!]$ such that $|e_i| \sqsubseteq t_3$. Hence by Theorem 5, $\exists e_3 \in LExp$ such that $e \to_l^* e_3$ and $t_3 \sqsubseteq |e_3|$, thus $|e_i| \sqsubseteq |e_3|$.
On the other hand to prove the converse implication let us assume some $t_1, t_2 \in [\![e]\!]$. Then by Theorem 5 we have that $\exists e_1, e_2 \in LExp$ such that $e \to_l^* e_1$, $t_1 \sqsubseteq |e_1|$ and $e \to_l^* e_2$, $t_2 \sqsubseteq |e_2|$. Hence by hypothesis we get some $e_3 \in LExp$ such that $e \to_l^* e_3$ and $|e_1| \sqsubseteq |e_3|$, $|e_2| \sqsubseteq |e_3|$. But then by Theorem 4 we have that $|e_3| \in [\![e]\!]$, with $t_i \sqsubseteq |e_i| \sqsubseteq |e_3|$. □

PROOF:[For Lemma 14] Given some $\theta_1, \theta_2 \in [\![\sigma]\!]$ we will define some $\theta \in [\![\sigma]\!]$ such that $\theta_1 \sqsubseteq \theta$ and $\theta_2 \sqsubseteq \theta$ as follows. For a given $X \in \mathcal{V}$ we know that $\theta_1(X), \theta_1(X) \in [\![\sigma(X)]\!]$ because $\theta_1, \theta_2 \in [\![\sigma]\!]$. But as $\sigma \in DSubst_\perp$ then $[\![\sigma(X)]\!]$ is a directed set—either because $X \in dom(\sigma)$ or because $X \notin dom(\sigma)$ thus $[\![\sigma(X)]\!] = [\![X]\!] = \{X, \perp\}$—, hence $\exists t \in [\![\sigma(X)]\!]$ such that $\theta_1(X) \sqsubseteq t$ and $\theta_2(X) \sqsubseteq t$. Therefore we choose $\theta(X) = t$. □

PROOF:[For Lemma 15]
By a case distintion over e:

- If $e \equiv X \in dom(\sigma)$: Then $\mathcal{P} \vdash_{CRWL} e\sigma \equiv \sigma(X) \twoheadrightarrow t$, so we can define:

$$\theta(Y) = \begin{cases} t & \text{if } Y \equiv X \\ \perp & \text{if } Y \in (dom(\sigma) \setminus \{X\}) \\ Y & \text{if } Y \notin dom(\sigma) \end{cases}$$

  Then $\theta \in [\![\sigma]\!]$ because obviously $\theta \in CSusbt_\perp$, and given $Z \in \mathcal{V}$.
  a) If $Z \equiv X$ then $\mathcal{P} \vdash_{CRWL} \sigma(Z) \equiv \sigma(X) \twoheadrightarrow t \equiv \theta(Z)$ by hypothesis.
  b) If $Z \in (dom(\sigma) \setminus \{X\})$ then $\mathcal{P} \vdash_{CRWL} \sigma(Z) \twoheadrightarrow \perp \equiv \theta(Z)$ by rule **B**.
  c) Otherwise $Z \notin dom(\sigma)$ and then $\mathcal{P} \vdash_{CRWL} \sigma(Z) \equiv Z \twoheadrightarrow Z \equiv \theta(Z)$ by rule **RR**.

  But then $\mathcal{P} \vdash_{CRWL} e\theta \equiv \theta(X) \equiv t \twoheadrightarrow t$ because $\forall t \in CTerm_\perp.\mathcal{P} \vdash_{CRWL} t \twoheadrightarrow t$ (a known property of *CRWL* which can be easily proved by induction on the structure of t).
- If $e \equiv X \notin dom(\sigma)$: Then given $\overline{Y} = dom(\sigma)$ it is easy to check that $[\overline{Y/\perp}] \in [\![\sigma]\!]$, so we can take $\theta = \{[\overline{Y/\perp}]\}$ for which $[\![e\sigma]\!] = [\![X\sigma]\!] = [\![X]\!] = [\![X[\overline{Y/\perp}]]\!] = [\![X\theta]\!]$.
- If $e \notin \mathcal{V}$ then we proceed by induction over the structure of $e\sigma \twoheadrightarrow t$:

  Base cases

  B Then $t \equiv \perp$, so given $\overline{Y} = dom(\sigma)$ we can take $\theta = \{[\overline{Y/\perp}]\}$ for which $e\theta \twoheadrightarrow \perp$ by **B**.

  RR Then $e \in \mathcal{V}$ and we are in the previous case.

  DC Similar to the case for $e \equiv X \notin dom(\sigma)$.

  Inductive steps

  DC Then $e \equiv c(e_1,\ldots,e_n)$, as $e \notin \mathcal{V}$, and we have:

$$\frac{e_1\sigma \twoheadrightarrow t_1\ \ldots\ e_n\sigma \twoheadrightarrow t_n}{e\sigma \equiv c(e_1\sigma,\ldots,e_n\sigma) \twoheadrightarrow c(t_1,\ldots,t_n) \equiv t}\ DC$$

  Then by IH or the proof of the other cases we have that $\forall i \in \{1,\ldots,n\}\ \exists\ \theta_i \in [\![\sigma]\!]$ such that $\mathcal{P} \vdash_{CRWL} e_i\theta_i \twoheadrightarrow t_i$. But as $\sigma \in DSusbt_\perp$ then we can apply Lemma 14 to obtain that $[\![e]\!]$ is a directed set, hence there must exists some $\theta \in [\![\sigma]\!]$ such that $\forall i \in \{1,\ldots,n\}\ \theta_i \sqsubseteq \theta$, and so by Lemma 1 we have $\forall i \in \{1,\ldots,n\}\ \mathcal{P} \vdash_{CRWL} e_i\theta \twoheadrightarrow t_i$, so we can build the following proof:

$$\frac{e_1\theta \twoheadrightarrow t_1\ \ldots e_n\theta \twoheadrightarrow t_n}{e\theta \equiv c(e_1\theta,\ldots,e_n\theta) \twoheadrightarrow c(t_1,\ldots,t_n) \equiv t}\ DC$$

  OR Very similar to the proof of the previuos case. We also have $e \equiv f(e_1,\ldots,e_n)$ (as $e \notin \mathcal{V}$) and given a proof for $e\sigma \equiv f(e_1,\ldots,e_n)\sigma \twoheadrightarrow t$, so we can apply the IH or the proof of the other cases to every $e_i\sigma \twoheadrightarrow p_i\mu$ to get some $\theta_i \in [\![\sigma]\!]$ such that $e_i\theta_i \twoheadrightarrow p_i\mu$. Then we can use lemmas 14 and 1 to use the obtained θ to compute the same values for the arguments of f, thus using the same substitution $\mu \in CSubst_\perp$ for parameter passing in OR.

□

PROOF:[For Theorem 10] As $CRWL^d$ inherits all the rules of *CRWL* then it is trivially complete. All that is left is proving that the rule $\mathbf{OR}^d$ is sound. Let us suppose an application of $\mathbf{OR}^d$ in which its premise is a *CRWL*-proof, not only a $CRWL^d$-proof, we will see that we can replace that application of $\mathbf{OR}^d$ with an application

of **OR**, obtaining exactly the same result. If the starting proof was the following:

$$\frac{r\sigma \twoheadrightarrow t}{f(p_1,\ldots,p_n)\sigma \twoheadrightarrow t}\ \text{OR}^d$$

with $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ and $\sigma \in DSubst_\perp$. Then, as σ is deterministic, applying Lemma 15 under $\mathcal{P} \vdash_{CRWL} r\sigma \twoheadrightarrow t$ we get that there must exist $\theta \in [\![\sigma]\!]$ such that $\mathcal{P} \vdash_{CRWL} r\theta \twoheadrightarrow t$. Besides, we can prove that $\forall i \in \{1,\ldots,n\},\ \mathcal{P} \vdash_{CRWL} p_i\sigma \twoheadrightarrow p_i\theta$, by induction on the structure of each p_i:

Base cases

• $p_i \equiv X \in \mathcal{V}$: Then $\mathcal{P} \vdash_{CRWL} p_i\sigma \equiv \sigma(X) \twoheadrightarrow \theta(X) \equiv p_i\theta$, as $\theta \in [\![\sigma]\!]$.

• $p_i \equiv c \in CS^0$: Then $\mathcal{P} \vdash_{CRWL} p_i\sigma \equiv c \twoheadrightarrow c \equiv p_i\theta$, by **DC**.

Inductive step Then $p_i \equiv c(t_1,\ldots,t_n)$ and we can do

$$\frac{\dfrac{IH}{t_1\sigma \twoheadrightarrow t_1\theta}\quad \ldots\quad \dfrac{IH}{t_n\sigma \twoheadrightarrow t_n\theta}}{c(t_1\sigma,\ldots,t_n\sigma) \twoheadrightarrow c(t_1\theta,\ldots,t_n\theta)}\ DC$$

As $\theta \in [\![\sigma]\!]$ then $\theta \in CSubst_\perp$ and so it can be used to apply **OR** as follows:

$$\frac{p_1\sigma \twoheadrightarrow p_1\theta\quad \ldots\quad p_n\sigma \twoheadrightarrow p_n\theta\quad r\theta \twoheadrightarrow t}{f(p_1,\ldots,p_n)\sigma \twoheadrightarrow t}\ OR$$

On the other hand, if the starting proof was:

$$\frac{r\sigma \twoheadrightarrow t}{f\sigma \equiv f \twoheadrightarrow t}\ \text{OR}^d \text{ with } (f \to r) \in \mathcal{P} \text{ y } \sigma \in DSubst_\perp$$

then we would have $\theta \in [\![\sigma]\!] \subseteq CSusbt_\perp$ such that $\mathcal{P} \vdash_{CRWL} r\theta \twoheadrightarrow t$, as in the previous case, and we could use it to apply **OR**:

$$\frac{r\theta \twoheadrightarrow t}{f\sigma \equiv f \twoheadrightarrow t}\ OR$$

We have just covered the case where the premise used to apply $\mathbf{OR}^d$ is also a *CRWL*-proof, but for any $CRWL^d$-proof we can apply this transformation from its leaves (the application of rules without premise, like **B** or **RR**) climbing to its parents (the proofs for which they are premises), obtaining an equivalent *CRWL*-proof. □

PROOF:[For Lemma 16] First we will prove this lemma for just one rewriting step. If the step was performed at the root of the expression then we have $e \equiv f(\overline{p})\sigma \to r\sigma \equiv e'$ for $(f(\overline{p}) \to r) \in \mathcal{P}$ and $\sigma \in Subst$. But as $\mathcal{P}$ is deterministic then $\sigma \in DSubst$, thus if $\mathcal{P} \vdash_{CRWL^d} r\sigma \to t$ then we could apply rule $\mathbf{OR}^d$, obtaining:

$$\frac{r\sigma \to t}{f(\overline{p})\sigma \to t}\ OR^d$$

But then $[\![e']\!]^d = [\![r\sigma]\!]^d \subseteq [\![f(\overline{p})\sigma]\!]^d = [\![e]\!]^d$, which combined with Theorem 10 yields $[\![e']\!] = [\![e']\!]^d \subseteq [\![e]\!]^d = [\![e]\!]$.

If the rewriting step was not performed at the root then we have $e \equiv s[a]_o \to s[a']_o \equiv e'$, where $a \to a'$ is performed at the root. But then by the previous case we have $[\![a']\!] \subseteq [\![a]\!]$, and so $[\![e']\!] = [\![s[a']_o]\!] \subseteq [\![s[a]_o]\!] = [\![e]\!]$, by the compositionality of *CRWL* (see [23] for details).

The extension of this proof to $e \to^* e'$ is a simple induction on the number of term rewriting steps. □

8.1.8 Rewriting and Call-time Choice: The HO case (Extended version)

Rewriting and Call-time choice: the HO case*

(Extended version)

Tech. Rep. UCM-SIC-3-08, 2008

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. It is known that the behavior of non-deterministic functions with call-time choice semantics, present in current functional logic languages, is not well described by usual approaches to reduction like ordinary term rewriting systems or λ-calculus. The presence of HO features makes things more difficult, since reasoning principles that are essential in a standard (i.e., deterministic) functional setting, like extensionality, become wrong. In this paper we propose *HOlet*-rewriting, a notion of rewriting with local bindings that turns out to be adequate for programs with HO non-deterministic functions, as it is shown by strong equivalence results with respect to *HOCRWL*, a previously existing semantic framework for such programs. In addition, we give a sound and complete notion of *HOlet*-narrowing, we show by a case study the usefulness of the achieved combination of semantic and reduction notions, and finally we prove within our framework that a standard approach to the implementation of HO features, namely translation to FO, is still valid for HO nondeterministic functions.

1 Introduction

Functional logic programming (FLP, for short; see [12, 14] for surveys) integrates features of logic programming and functional programming. Typically FLP adopts mostly a (lazy) functional style, thus making intensive use of higher order (HO) functions. However, most of the work about FLP focuses on first order (FO) aspects of programs, thus limiting the applicability of results.

This is not a satisfactory situation, especially taking into account that the presence of functions that are at the same time HO and non-deterministic leads to somehow surprising behaviors, as shown by the example we sent recently to the Curry mailing list [13]:

Example 1. Consider the following program computing with natural numbers represented by the constructors 0 and $s/1$, and where + is defined as usual.

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03) and Promesas-CAM (S-0505/TIC/0407).

```
g X -> 0        f -> g          f' X -> f X
h X -> s 0      f -> h

fadd F G X -> (F X) + (G X)     fdouble F -> fadd F F
```

Notice that f and f' are non-deterministic functions that are (by definition of f') extensionally equivalent; from the point of view of standard functional programming they should be seen as 'the same function'. However, consider the expressions *(fdouble f 0)* and *(fdouble f' 0)*. In modern FLP languages like Curry [16] or Toy [20], the possible values for *(fdouble f 0)* are *0, s (s 0)*, while *(fdouble f' 0)* can be in addition reduced to *s 0*.

This behavior corresponds to *call-time choice* [17, 11], the semantics for non-determinism adopted by those systems. Operationally call-time choice is very close to the *sharing* mechanism used in functional languages to implement lazy evaluation.

The example was sent[1] to point out that η-expansion and η-reduction are not valid for such systems, because extensionally equivalent functions (e.g., f and f') can be semantically distinguishable when put in the same context (e.g., *double* [ ] 0), a fact that does not happen neither in standard (i.e, deterministic) functional programs[2], nor in FO FLP. We remark also that with *run-time choice* [17, 11], f and f' will be indistinguishable (*double f 0* and *double f' 0* would both produce *0, s 0, s (s 0)* as possible results). Therefore, it is the combination *HO + Non-determinism + call-time choice* which makes things different.

That combination was addressed in *HOCRWL* [7, 8], an extension to HO of *CRWL* [3] [11], a semantic framework specifically devised for FLP with call-time choice semantics for non-determinism (see [27] for a survey of *CRWL* and its extensions). *HOCRWL* provides logic and model-theoretic semantics, based on an *intensional* view of functions, where different descriptions –in the form of *HO-patterns*– of the same extensional function are distinguished as different data. This allows expressive programs and is simpler than λ-calculus-based HO unification, which is an alternative approach followed in the logic programming setting [22]. Previous work on the intensional view of HO-FLP [10] did not consider non-determinism. Other works covering HO in FLP, [23, 15], consider orthogonal or inductively sequential (henceforth deterministic) systems; if extended directly to the non-deterministic case, they would realize run-time choice, as happens also with [4], where a type-based translation to FO in the spirit of [28, 9] is proposed. We remark also that [15] is close to the theory of HO rewriting [26], and therefore has η-expansion as a valid procedure, against the expected properties of the languages considered by ours. Finally, [1] copes with call-time choice but their approach to HO is again based on a FO-translation, in contrast to ours.

[1] As far as we know, it was the first time that this behavior was noticed.
[2] Although the addition of primitive functions not definable in the language like *seq* in Haskell [24] can also destroy extensionality.
[3] CRWL stands for *Constructor Based Rewriting Logic*.

A weak point of the original *(HO)CRWL*-way to FLP is that it does not come with a clear, simple notion of one-step reduction similar to one-step rewriting. In [19] we proposed *let-rewriting*, a notion of rewriting with local bindings adequate to FO *CRWL* semantics, and at the same time simpler and more abstract than other reduction notions based on term graph rewriting [25, 6] or natural operational semantics [1]. *Let*-rewriting was generalized to *let*-narrowing in [18].

Our aim in this work is to extend the notion of *let*-rewriting/narrowing to the HO case. We address various foundational aspects –definition of *HOlet*-rewriting and equivalence wrt the declarative semantics given by *HOCRWL* (Sect. 3), *HOlet*-narrowing and its soundness and completeness wrt *HOlet*-rewriting (Sect. 4)– and also more applied aspects, as are the use of our framework to language development (Sect. 5) or the proof of correctness within our framework of a scheme of translation to FO, the basis of a standard approach [28, 9, 4] to the implementation of HO stuff in FO settings.

There are still some other important issues –evaluation strategies (including concurrency), types, constraints– that have been left out of the scope of the paper. Finally, we are not inventing HO FLP, but only contributing to some aspects of its foundation. Therefore it is not our aim in this paper convincing of the practical interest of HO FLP: other documents [16, 27, 7, 4] contain enough evidences of that. Proofs can be found in an appendix.

2 Preliminaries: *HOCRWL*

We present here some basic notions and new results about *HOCRWL* [7].

2.1 Expressions, patterns and programs

We consider *function* symbols $f, g, \ldots \in FS$, *constructor* symbols $c, d, \ldots \in CS$, and *variables* $X, Y, \ldots \in \mathcal{V}$; each $h \in FS \cup CS$ has an associated *arity*, $ar(h) \in \mathbb{N}$; FS^n (resp. CS^n) is the set of function (resp. constructor) symbols with arity n. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects o. The set of *applicative expressions* is defined by $Exp \ni e ::= X \mid h \mid (e_1\ e_2)$. As usual, application is left associative and outer parentheses can be omitted, so that $e_1\ e_2\ \ldots e_n$ stands for $((\ldots(e_1\ e_2)\ldots)\ e_n)$. The set of variables occurring in e is written by $var(e)$. A distinguished set of expressions is that of *patterns* $t, s \in Pat$, defined by: $t ::= X \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_m$, where $0 \leq n \leq ar(c), 0 \leq m < ar(f)$. Patterns are irreducible expressions playing the role of *values*. *FO-patterns*, defined by $FOPat \ni t ::= X \mid c\ t_1 \ldots t_n\ (n = ar(c))$, correspond to FO constructor terms, representing ordinary non-functional data-values. Partial applications of symbols $h \in FS \cup CS$ to other patterns are HO-patterns and can be seen as truly data-values representing functions from an *intensional* point of view. Examples of patterns with the signature of Ex. 1 are: *0, s X, s, f', fadd f' f'*. The last three are HO-patterns. Notice that *f, fadd f f* are not patterns since f is not a pattern ($ar(f) = 0$).

Expressions $X\ e_1 \ldots e_m$ ($m \geq 0$) are called *flexible* (*variable application* when $m > 0$). *Rigid* expressions have the form $h\ e_1 \ldots e_m$; moreover, they are *junk* if $h \in CS^n$ and $m > n$, *active* if $h \in FS^n$ and $m \geq n$, and *passive* otherwise.

Contexts are expressions with a hole defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$. Application of $\mathcal{C}$ to e (written $\mathcal{C}[e]$) is defined by $[\,][e] = e$; $(\mathcal{C}\ e')[e] = \mathcal{C}[e]\ e'$; $(e'\ \mathcal{C})[e] = e'\ \mathcal{C}[e]$. Substitutions $\theta \in Subst$ are finite mappings from variables to expressions; $[X_i/e_i, \ldots, X_n/e_n]$ is the substitution which assigns $e_i \in Exp$ to the corresponding $X_i \in \mathcal{V}$. We will mostly use *pattern-substitutions* $PSubst = \{\theta \in Subst \mid \theta(X) \in Pat, \forall X \in \mathcal{V}\}$. We write ϵ for the identity substitution, $dom(\theta)$ for the domain of θ, and $vRan(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$.

As usual while describing semantics of non-strict languages, we enlarge the signature with a new 0-ary constructor symbol $\bot$, which can be used to build the sets $Expr_\bot, Pat_\bot, PSubst_\bot$ of *partial* expressions, patterns and p-substitutions resp. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

A *HOCRWL*-program (or simply a *program*) consists of one or more *program rules* for each $f \in FS^n$, having the form $f\ t_1 \ldots t_n \to r$ where $(t_1, \ldots, t_n)$ is a linear (i.e. variables occur only once) tuple of (maybe HO) patterns and r is any expression. Notice that confluence or termination is not required, and that r may have variables not occurring in $f\ t_1 \ldots t_n$ (we write $vExtra(R)$ for such variables in a rule R). The original *HOCRWL* logic considered also *joinability* conditions in rules to achieve a better treatment of strict equality as built-in, which is a subject orthogonal to the aims of this paper. Therefore, we consider only unconditional rules.

Some related languages, like Curry, do not allow HO-patterns in left-hand sides of function definitions. We remark that all the notions and results in the paper are applicable to programs with this restriction and we stress the fact that Example 1 is one of them.

Given a program $\mathcal{P}$, the set of its rule instances is $[\mathcal{P}] = \{(l \to r)\theta \mid (l \to r) \in \mathcal{P}, \theta \in PSubst\}$. The set $[\mathcal{P}]_\bot$ is defined similarly replacing $PSubst$ by $PSubst_\bot$. To require $\theta \in PSubst_{(\bot)}$ instead of $\theta \in Subst_{(\bot)}$ is essential to achieve call-time choice in the next sections.

2.2 The *HOCRWL* proof calculus [7]

The semantics of a program $\mathcal{P}$ is determined in *HOCRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in Pat_\bot$, meaning informally that t is (or approximates to) a possible value of e, obtained by evaluation of e using $\mathcal{P}$ under call-time choice. Besides this logical semantics, *HOCRWL* programs come in [7] with a model-theoretic semantics based on applicative algebras, with existence of a least Herbrand model. We will not use this aspect of the semantics here.

The *HOCRWL*-proof calculus is presented in Fig. 1. We write $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in that calculus using the program $\mathcal{P}$.

The *HOCRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}}_{HOCRWL} = \{t \in Pat_\bot \mid \mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t\}$. $\mathcal{P}$ and *HOCRWL* are frequently omitted in those notations.

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(RR)}\ \frac{}{x \twoheadrightarrow x}\quad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1\ \ldots\ e_n \twoheadrightarrow t_m}{h\ e_1 \ldots e_m \twoheadrightarrow h\ t_1 \ldots t_m} \quad h \in \Sigma,\ \text{if } h\ t_1 \ldots t_m \text{ is a partial pattern, } m \geq 0$$

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow t_1 \ldots\ e_n \twoheadrightarrow t_n\ \ r\ a_1 \ldots a_m \twoheadrightarrow t}{f\ e_1 \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t} \quad \text{if } m \geq 0, (f\ t_1 \ldots t_n \to r) \in [\mathcal{P}]_\bot$$

Fig. 1. (*HOCRWL*-calculus)

In Example 1 we have $[\![f double\ f\ 0]\!] = \{0, s\ (s\ 0), \bot, s\ \ \bot, s\ (s\ \ \bot)\}$ and $[\![f double\ f'\ 0]\!] = \{0, s\ 0, s\ (s\ 0), \bot, s\ \ \bot, s\ (s\ \ \bot)\}$.

We will use the following (new) result stating an important compositionality property of the semantics of *HOCRWL*-expressions: the semantics of a whole expression depends only on the semantics of its constituents, in a particular form reflecting the idea of call-time choice. The second part of the theorem is a technical result, needed in some proofs, concerning the size of the involved derivations.

Theorem 1 (Compositionality of *HOCRWL* semantics).

(i) $[\![\mathcal{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathcal{C}[t]]\!]$, *for any program* $\mathcal{P}$ *and expression* $e \in Exp_\bot$.
In other terms, $\mathcal{C}[e] \twoheadrightarrow t \Leftrightarrow \exists s.(e \twoheadrightarrow s \wedge \mathcal{C}[s] \twoheadrightarrow t)$.

(ii) In the ($\Rightarrow$) *part of (i), if* $t \neq \bot, \mathcal{C} \neq [\ ]$ *and the derivation of* $\mathcal{C}[e] \twoheadrightarrow t$ *has size* K*, then the derivations of* $e \twoheadrightarrow s$ *and* $\mathcal{C}[s] \twoheadrightarrow t$ *can be chosen with sizes* $< K$ *and* $\leq K$ *respectively.*

3 Higher order *let*-rewriting

To express sharing, as is required for call-time choice, we enhance the syntax of expressions (and contexts) with a *let* construct for local bindings, in the spirit of [5, 21, 19]: $LExp \ni e ::= X \mid h \mid e_1\ e_2 \mid let\ X = e_1\ in\ e_2$
$Cntxt \ni \mathcal{C} ::= [\ ] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$
We consider expressions $let\ X = e_1\ in\ e_2$ as passive and rigid. The sets $FV(e)$ and $BV(e)$ of free and bound variables resp. of a *let*-expression e are defined as:

$$FV(X) = \{X\};\ FV(h\ \overline{e}) = \bigcup_{e_i \in \overline{e}} FV(e_i);$$
$$FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\});$$
$$BV(X) = \emptyset;\ BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i);$$
$$BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\}$$

Notice that with the given definition of $FV(let\ X = e_1\ in\ e_2)$ recursive *let*-bindings are not allowed since the possible occurrences of X in e_1 are not considered as bound and therefore refer to a 'different' X. We assume appropriate renamings of bound variables ensuring that bound and free variables are kept distinct, and that whenever θ is applied to $e \in LExp$, $BV(e) \cap (dom(\theta) \cup vRan(\theta)) = \emptyset$, so that $(let\ X = e_1\ in\ e_2)\theta = let\ X = e_1\theta\ in\ e_2\theta$ and $(\mathcal{C}[e])\theta = \mathcal{C}\theta[e\theta]$.

The *shell* of an expression, written as $|e|$, is a pattern containing the 'stable' outer information of e, not to be destroyed by reduction:

$$|X\ e_1 \dots e_m| = \begin{cases} X & \text{if } m = 0 \\ \bot & \text{if } m > 0 \end{cases}$$

$$|h\ e_1 \dots e_m| = \begin{cases} h\ |e_1| \dots |e_m| & \text{if } (h \in CS^n, m \leq n) \text{ or } (h \in FS^n, m < n) \\ \bot & \text{otherwise (junk or active expression)} \end{cases}$$

$$|(let\ X = e_1\ in\ e_2)\ a_1 \dots a_m| = |(e_2[X/e_1])\ a_1 \dots a_m|$$

Notice that in FO [19] we defined $|(let\ X = e_1\ in\ e_2)| = |e_2|[X/|e_1|]$. This would lose information in the HO case: for instance, $|let\ X = s\ in\ X\ 0|$ would be $\bot$, instead of the more accurate $s\ 0$ given by the definition above.

The $HOCRWL_{let}$ proof calculus for proving statements $e \twoheadrightarrow t$ ($e \in LExp_\bot, t \in Pat_\bot$) results from adding to Fig. 1 the rule:

$$\textbf{(Let)} \quad \frac{e_1 \twoheadrightarrow t_1 \quad (e_2[X/t_1])\ a_1 \dots a_m \twoheadrightarrow t}{(let\ X = e_1\ in\ e_2)\ a_1 \dots a_m \twoheadrightarrow t} \qquad (m \geq 0)$$

It is easy to see that for programs and expressions without *let*s both calculi coincide, giving $[\![e]\!]_{HOCRWL} = [\![e]\!]_{HOCRWL_{let}}$, and then we write simply $[\![e]\!]$.

Theorem 1 does not hold as it is for *let*-expressions (assume, for instance, the program rule $f\ 0 = 1$ and take $e \equiv f\ X$, $\mathcal{C} \equiv let\ X{=}0\ in\ [\ ]$). However, a more limited form of compositionality will suffice to our needs:

Theorem 2 (Weak compositionality of $HOCRWL_{let}$ semantics). *For any $\mathcal{P}$ and $e, e' \in LExp_\bot$: $[\![C\ [e]]\!] = \bigcup_{t \in [\![e]\!]} [\![C\ [t]]\!]$, if $BV(C) \cap FV(e) = \emptyset$.*

As a consequence, (i) $[\![e\ e']\!] = \bigcup_{t \in [\![e]\!]} [\![t\ e']\!]$ *(ii)* $[\![e\ e']\!] = \bigcup_{t \in [\![e']\!]} [\![e\ t]\!]$

(iii) $[\![let\ X = e\ in\ e']\!] = \bigcup_{t \in [\![e]\!]} [\![e'[X/t]]\!]$

3.1 Rewriting with local bindings

Figure 2 defines the *HOlet*-rewriting relation $\rightarrow^l$. Rule *(Fapp)* uses a program rule to reduce a function application, but only when the arguments are already patterns, otherwise call-time choice would be violated. Non-pattern arguments of applications are moved to local bindings by *(LetIn)*. Local bindings of patterns to variables are applied in *(Bind)*, since in this case copying is harmless. *(Elim)* erases useless bindings. *(Flat)* and *(LetAp)* manage local bindings; they are needed to avoid some reductions to get stuck. Notice that with the variable convention, the condition $Y \notin FV(e_3)$ in *(Flat)* and *(LetAp)* would not be needed; we have written it in order to keep the rules independent of the convention. Finally, any of these rules can be applied to any subexpression by *(Contx)*.

It includes an additional technical condition to avoid undesired variable captures when *(Fapp)* was applied inside a surrounding context and the used program rule has extra variables. If, for instance, a program rule is $f \to Y$, the rule *(Contxt)* avoids the step *let X=0 in f* $\to^l$ *let X=0 in X* and also the step *let X=f in X* $\to^l$ *let X=X in X*.

(Fapp) $f\ t_1 \ldots t_n \to^l r$, if $(f\ t_1 \ldots t_n \to r) \in [\mathcal{P}]$

(LetIn) $e_1\ e_2 \to^l let\ X = e_2\ in\ e_1\ X$ (X fresh), if e_2 is an active expression, variable application, junk or *let* rooted expression.

(Bind) $let\ X = t\ in\ e \to^l e[X/t]$, if $t \in Pat$

(Elim) $let\ X = e_1\ in\ e_2 \to^l e_2$, if $X \notin FV(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to^l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \to^l let\ X = e_1\ in\ e_2\ e_3$, if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \to^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \to^l e'$ using any of the previous rules, and in case $e \to^l e'$ is a (Fapp) step using $(f\ \overline{p} \to r)\theta \in [\mathcal{P}]$ then $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$.

Fig. 2. Higher order *let*-rewriting relation $\to^l$

The following derivation corresponds to Example 1:

fdouble f 0 $\to^l_{\{LetIn,Cntx\}}$ *(let F=f in fdouble F) 0*
$\to^l_{LetAp}$ *let F=f in fdouble F 0* $\to^l_{\{Fapp,Cntx\}}$ *let F=f in fadd F F 0*
$\to^l_{\{Fapp,Cntx\}}$ *let F=f in F 0 + F 0*
$\to^l_{\{Fapp,Cntx\}}$ *let F=g in F 0 + F 0* $\to^l_{Bind}$ *g 0 + g 0* $\to^{l^*}$ *0*

Notice that the fist step is justified because f is active. In contrast, since f' is a pattern, a derivation for *fdouble f' 0* could proceed as follows:

fdouble f' 0 $\to^l$ *fadd f' f' 0* $\to^l$ *f' 0 + f' 0* $\to^{l^*}$ *f 0 + f 0* $\to^{l^*}$ *g 0 + h 0* $\to^{l^*}$ *s 0*

The rules of $\to^l$ have been carefully tuned up to ensure that program rules are the only possible source of non-termination, as ensured by the following result.

Proposition 1. *The relation* $\to^l_{\setminus Fapp}$ *defined by the rules of Fig. 2 except* (Fapp) *is terminating.*

This is a natural requirement. However, at some point we will find useful to consider the more liberal relation $\to^L$ obtained replacing *(LetIn)* by:

(LetIn') $e_1\ e_2 \to^L let\ X = e_2\ in\ e_1\ X$ (X fresh)

which is less restrictive (then $\to^l \subseteq \to^L$). However $\to^L_{\setminus Fapp}$ becomes non-terminating, as shown by: $s\ 0 \to^l_{LetIn'} let\ X = 0\ in\ s\ X \to^l_{Bind} s\ 0 \to^l \ldots$

3.2 Adequacy of *HOlet*-rewriting to ***HOCRWL***

We compare here $\rightarrow^l$ to *HOCRWL*-derivability $\twoheadrightarrow$, proving that essentially $\rightarrow^l$ gives no more (*soundness*) and no less (*completeness*) results than $\twoheadrightarrow$.

As in [19], the following notion is useful to establish soundness:

Definition 1 (Hypersemantics).

(i) The hypersemantics *of an expression $e \in LExp_\bot$, written as $[\![e]\!]$, is a mapping $[\![e]\!] : PSubst_\bot \longrightarrow \mathcal{P}(Pat_\bot)$ defined by $[\![e]\!](\theta) = [\![e\theta]\!]$.*
(ii) Hypersemantics of expressions are ordered as follows:

$$[\![e_1]\!] \Subset [\![e_2]\!] \text{ iff } [\![e_1\theta]\!] \subseteq [\![e_2\theta]\!],\ \forall \theta \in PSubst_\bot$$

The main reason for introducing hypersemantics is that it enjoys the following nice monotonicity-under-contexts property, while $[\![\_]\!]$ does not:

Lemma 1 (Monotonicity of hypersemantics).
$[\![e]\!] \Subset [\![e']\!]$ implies $[\![\mathcal{C}[e]]\!] \Subset [\![\mathcal{C}[e']]\!]$, for any $e, e' \in LExp_\bot$, $\mathcal{C} \in Cntxt$.

Monotonicity under contexts is the key for our next result, stating that hypersemantics does not grow under *HOlet*-rewriting steps:

Lemma 2 (One-Step Hyper-Soundness of *HOlet*-rewriting).
$e \rightarrow^l e'$ implies $[\![e']\!] \Subset [\![e]\!]$, for any $e, e' \in LExp$.

Notice that $\Subset$ cannot be replaced here by $=$, due to non-determinism.

Lemma 2, together with the easy observation that $[\![e_1]\!] \Subset [\![e_2]\!]$ implies $[\![e_1]\!] \subseteq [\![e_2]\!]$ (just take $\theta = \epsilon$) and an obvious induction over derivation lengths, leads to our main correctness result for $\rightarrow^l$:

Theorem 3 (Soundness of *HOlet*-rewriting). *Let $\mathcal{P}$ be a program, $e, e' \in LExp$. Then: (i) $e \rightarrow^{l^*} e'$ implies $[\![e']\!] \subseteq [\![e]\!]$, and therefore $e \twoheadrightarrow |e'|$*
(ii) $e \rightarrow^{l^*} t$ implies $e \twoheadrightarrow t$, for any $t \in Pat$.

The proof of this result can be easily extended to the larger relation $\rightarrow^L$ (the one which uses (LetIn') instead of (LetIn)).

Regarding completeness of *let*-rewriting, a key in the FO case was the *peeling lemma* ([19], Lemma 7), a technical result giving a kind of standard form in which the implicit or explicit sharing information contained in $e \in Exp$ can be expressed. It is not obvious how to proceed in the HO case, since straightforward generalizations of the FO peeling lemma turn out to be false. However, we have found that the following weak HO version is enough for our purposes:

Lemma 3 (Weak peeling lemma). *Let $h\ e_1 \ldots e_m \in Exp$ with $h \in \Sigma^n$ (n and m can be different). Then $h\ e_1 \ldots e_m \rightarrow^{l^*} let\ \overline{X = a}\ in\ h\ t_1 \ldots t_m$, for some $t_1, \ldots, t_m \in Pat, \overline{a} \subseteq Exp$ such that $|\overline{a}| = \overline{\bot}, t_i \equiv e_i$ for every $e_i \in Pat$. Besides, in this derivation the rule (Fapp) is not applied.*

With this result and some monotonicity properties of *HOCRWL*-derivability, we can prove a very technical but strong completeness result for $\rightarrow^l$ wrt $\twoheadrightarrow$:

Lemma 4 (Completeness lemma for *HOlet*-rewriting). *For any program $\mathcal{P}$, $e \in Exp$ and $t \in Pat_\bot$ with $t \neq \bot$, the following holds: $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ implies $e \rightarrow^{l^*} let\ \overline{X = a}\ in\ t'$, for some $t' \in Pat$ and $\overline{a} \subseteq Exp$ in such a way that $t \sqsubseteq |let\ \overline{X = a}\ in\ t'|$ and $|a_i| = \bot$ for all $a_i \in \overline{a}$. As a consequence, $t \sqsubseteq t'[\overline{X/\ \bot}]$.*

The condition $t \neq \bot$ is needed, as can be seen just taking $\mathcal{P} = \{f \rightarrow f\}$, $e \equiv f$ and $t \equiv \bot$.

From Lemma 4 we can obtain our main completeness result for $\rightarrow^l$:

Theorem 4 (Completeness of *HOlet*-rewriting). *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in Pat_\bot$. Then:*
(i) $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ implies $e \rightarrow^{l^*} e'$, for some $e' \in LExp$ such that $t \sqsubseteq |e'|$.
(ii) If in addition $t \in Pat$, then $e \rightarrow^{l^*} t$.

Joining together the last parts of Theorems 3 and 4, we obtain a strong equivalence result for $\rightarrow^l$ and $\twoheadrightarrow$:

Theorem 5 (Equivalence of *HOlet*-rewriting and *HOCRWL*).
$\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t$ iff $e \rightarrow^{l^*} t$, for any $\mathcal{P}$, $e \in Exp$, and $t \in Pat$.

This justifies our claim that $\rightarrow^l$ is truly the reduction face of *HOCRWL*-semantics.

4 Higher order *let*-narrowing

For some FLP computations rewriting is not enough, and must be lifted to some kind of *narrowing*; this happens when the expression being reduced contains variables for which different bindings might produce different evaluation results. Narrowing is an old subject in the fields of theorem proving and declarative programming. Since classical rewriting is not correct for call-time choice, classical narrowing cannot be either (because rewriting is a particular case of narrowing). In [18] we proposed a notion of narrowing adequate to FO *let*-rewriting, and now we extend it to HO. As happens in [7, 4], *HOlet*-narrowing may bind variables to HO-patterns.

Figure 3 contains the rules for the one-step *HOlet*-narrowing relation $e \leadsto^l_\theta e'$, expressing that e is narrowed to e' producing the substitution $\theta \in PSubst$. In *(X)* we collect those cases of *HOlet*-rewriting corresponding also to narrowing steps with empty substitution. *(Narr)* is the proper rule of narrowing for function application; it may produce HO bindings if the used program rule has HO patterns. Notice that, for the sake of generality, we do not require that θ is a mgu. *(VAct)* and *(VBind)* are rules producing HO bindings for flexible expressions (or subexpressions, in the case of *(VBind)*). We have preferred this pair of rules instead of the rule

(VNarr) $X\ e \leadsto^L_{[X/t]} t\ (e[X/t])$, for any $t \in Pat$

which is simpler, but also 'wilder' because it creates a larger search space. Finally, (Contxt) is a contextual rule where, as in [18], it is crucial to protect bound variables from narrowing (condition (i)) and to avoid variable capture (condition (ii), automatically fulfilled if mgu's are used in *(Narr)* and *(VAct)*, and fresh *shallow* patterns –i.e., of the form $h\ X_1 \dots X_n$– in *(VBind)*).

(X) $e \leadsto^l_\epsilon e'$ if $e \to^l e'$ using $\boldsymbol{X} \in \{Elim, Bind, Flat, LetIn, LetAp\}$ in Figure 2.

(Narr) $f\ \bar{t} \leadsto^l_\theta r\theta$, for any fresh variant $(f\ \bar{p} \to r) \in \mathcal{P}$ and $\theta \in PSubst$ such that $f\ \bar{t}\theta \equiv f\ \bar{p}\theta$.

(VAct) $X\ t_1 \dots t_k \leadsto^l_\theta r\theta$, if $k > 0$, for any fresh variant $(f\ \bar{p} \to r) \in \mathcal{P}$ and $\theta \in PSubst$ such that $(X\ t_1 \dots t_k)\theta \equiv f\ \bar{p}\theta$.

(VBind) $let\ X = e_1\ in\ e_2 \leadsto^l_\theta e_2\theta[X/e_1\theta]$, if $e_1 \notin Pat$, for any $\theta \in PSubst$ that makes $e_1\theta \in Pat$, provided that $X \notin (dom(\theta) \cup vRan(\theta))$.

(Contx) $\mathcal{C}[e] \leadsto^l_\theta \mathcal{C}\theta[e']$ for $\mathcal{C} \neq [\,]$, if $e \leadsto^l_\theta e'$ by any of the previous rules, and the following conditions hold:
- i) $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$
- ii) • If the step is *(Narr)* or *(VAct)* using $(f\ \bar{p} \to r) \in \mathcal{P}$, then $vRan(\theta|_{\setminus var(\bar{p})}) \cap BV(\mathcal{C}) = \emptyset$
  • If the step is *(VBind)* then $vRan(\theta) \cap BV(\mathcal{C}) = \emptyset$

Fig. 3. Higher order *let*-narrowing calculus $\leadsto^l$

Taking Example 1, a narrowing derivation for *fdouble F 0* would start with some *(X)* 'rewriting' steps:

$fdouble\ F\ 0 \leadsto^l_\epsilon fadd\ F\ F\ 0 \leadsto^l_\epsilon F\ 0 + F\ 0 \leadsto^l_\epsilon let\ X{=}F\ 0\ in\ X + F\ 0$

At this point, notice first that we cannot narrow on X, because it is a bound variable. Instead, we can apply *(VAct+Contx)*:

$$let\ X{=}F\ 0\ in\ X + F\ 0 \leadsto^l_{\{F/g\}} let\ X{=}0\ in\ X + g\ 0 \leadsto^{l*}_\epsilon 0$$

Other similar derivations using *(VAct+Contx)* would bind F to h (with final result *s (s 0)*), or to f' (with possible results *0, s 0, s (s 0)*). Notice that the binding X/f is not legal, since f is not a pattern.

Alternatively we could have applied *(VBind)*, obtaining:

$$let\ X{=}F\ 0\ in\ X + F\ 0 \leadsto^l_{\{F/s\}} s\ 0 + s\ 0 \leadsto^{l*}_\epsilon s\ (s\ 0)$$

We remark that, in our untyped framework, other 'ill-typed' bindings could be tried, like $F/fadd\ 0$ or $F/fdouble$. This is a symptom of known problems [4, 8] of the interaction with types of the intensional view of HO, that are partially alleviated in [4] by a typed version of a FO translation (see Sect. 6), but in general require (see [8]) bringing types to computations, a problem yet not well solved in practice. All these type-related issues are out of the scope of the paper.

A basic fact about completeness of *let*-narrowing in the FO case was that $e \leadsto^{l*}_\theta e'$ implied $e\theta \to^{l*} e'$, $\forall \theta \in CSubst$, which is closely related to the fact that FO *let*-rewriting is closed under c-substitutions. None of both facts hold with

HO $\rightsquigarrow^l$, $\rightarrow^l$ and $\theta \in PSubst$: consider for instance $e \equiv s\ (Y\ 0) \rightarrow^l let\ X = Y\ 0\ in\ s\ X \equiv e'$ and $\theta = [Y/s]$, for which $e\theta \equiv s\ (s\ 0) \not\rightarrow^l let\ X = s\ 0\ in\ s\ X \equiv e'\theta$. Similarly, we have $e \equiv s\ (Y\ 0) \rightsquigarrow^L_\epsilon let\ X = Y\ 0\ in\ s\ X \rightsquigarrow^L_{[Y/s]} let\ X = s\ 0\ in\ s\ X \equiv e'$, but $e\theta \equiv s\ (s\ 0) \not\rightarrow^l e'$.

At this point the relation $\rightarrow^L$ of Sect. 3 becomes useful, because we have:

Lemma 5 (Closedness of $\rightarrow^L$ under $PSubst$). *For every $e, e' \in LExp, \theta \in PSubst$, $e \rightarrow^{L^*} e'$ implies $e\theta \rightarrow^{L^*} e'\theta$.*

Now we can prove soundness of HO *let*-narrowing wrt. $\rightarrow^L$:

Theorem 6 (Soundness or $\rightsquigarrow^l$ wrt $\rightarrow^L$). *For any $e, e' \in LExp$, $e \rightsquigarrow^{l^*}_\theta e'$ implies $e\theta \rightarrow^{L^*} e'$.*

And now, taking into account Th. 3 (which holds also for $\rightarrow^L$), we get:

Theorem 7 (Soundness of *let*-narrowing). *For any $e, e' \in LExp$, $t \in Pat$:*

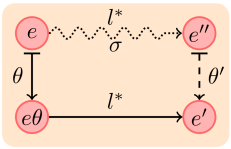
a) If $e \rightsquigarrow^{l^*}_\theta e'$ then $[\![e']\!] \subseteq [\![e\theta]\!]$ *b) If $e \rightsquigarrow^{l^*}_\theta t$ then $e\theta \rightarrow^{l^*} t$*

Regarding completeness, the following lemma shows how we can lift any $\rightarrow^l$ derivation to a $\rightsquigarrow^l$ derivation. This is surely the most involved result in the paper.

Lemma 6 (Lifting lemma for *HOlet*-rewriting). *Let $e, e' \in LExp$ such that $e\theta \rightarrow^{l^*} e'$ for some $\theta \in PSubst$, and let $\mathcal{W}, \mathcal{B} \subseteq \mathcal{V}$ with $dom(\theta) \cup FV(e) \subseteq \mathcal{W}$, $BV(e) \subseteq \mathcal{B}$ and $(dom(\theta) \cup vRan(\theta)) \cap \mathcal{B} = \emptyset$, and for each instance of a program rule $R\gamma \in [\mathcal{P}]$ used in an (Fapp) step of $e\theta \rightarrow^{l^*} e'$ then $vRan(\gamma|_{vExtra(R)}) \cap \mathcal{B} = \emptyset$. Then there exist a derivation $e \rightsquigarrow^{l^*}_\sigma e''$ and $\theta' \in PSubst$ such that:*

$$(i)\ e''\theta' = e' \qquad (ii)\ \sigma\theta' = \theta[\mathcal{W}] \qquad (iii)\ (dom(\theta') \cup vRan(\theta')) \cap \mathcal{B} = \emptyset$$

Besides, the HOlet*-narrowing derivation can be chosen to use mgu's at each* **(Narr)** *or* **(VAct)** *step, and fresh shallow patterns in the range for each* **(VBind)** *step. Graphically:*



With the aid of this lemma we can reach our completeness result for $\rightsquigarrow^l$:

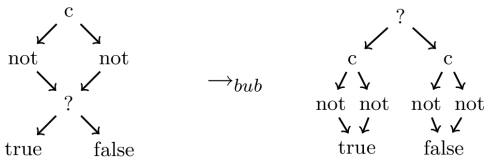
Theorem 8 (Completeness of *HOlet*-narrowing wrt. *HOlet*-rewriting). *Let $e, e' \in LExp$ and $\theta \in PSubst$. If $e\theta \rightarrow^{l^*} e'$, then there exist a* HOlet*-narrowing derivation $e \rightsquigarrow^{l^*}_\sigma e''$ and $\theta' \in PSubst$ such that $e''\theta' \equiv e'$ and $\sigma\theta' = \theta[FV(e)]$.*

5 A case of study: correctness of bubbling

Having equivalent notions of semantics and reduction allows to reason interchangeably at the rewriting and the semantic levels. We demonstrate the power of such technique by a case study where *let*-rewriting provides a good level of abstraction to formulate a new operational rule (*bubbling*), while the semantic point of view is appropriate for proving its correctness.

Bubbling, proposed in [3], is an operational rule devised to improve the efficiency of functional logic computations. Its correctness was formally studied in [2] in the framework of a variant [6] of term graph rewriting.

The idea of bubbling is to concentrate all non-determinism of a system into a *choice* operation ? defined by the rules $X\ ?\ Y \to X$ *and* $X\ ?\ Y \to Y$, and to lift applications of ? out of a surrounding context, as illustrated by the following graph transformation taken from [2]:



As it is shown in [3], bubbling can be implemented in such a way that many functional logic programs become more efficient, but we will not deal with these issues here.

Due to the technical particularities of term graph rewriting, not only the proof of correctness, but even the definition of bubbling in [3, 2] are involved and need subtle care concerning the appropriate contexts over which choices can be bubbled. In contrast, bubbling can be expressed within our framework (moreover, generalized to HO) in a remarkably easy and abstract way as a new rewriting rule: **(Bub)** $\mathcal{C}[e_1?e_2] \to^{bub} \mathcal{C}[e_1]?\mathcal{C}[e_2]$, for $e_1, e_2 \in LExp$

With this rule, the bubbling step corresponding to the graph transformation of the example above is: *let X = true ? false in c (not X) (not X)* $\to^{bub}$ *let X = true in c (not X) (not X) ? let X = false in c (not X) (not X)*

Notice that the effect of this bubbling step is not a shortening of any existing *HOlet*-rewriting derivation; bubbling is indeed a genuine new rule, the correctness of which must be therefore subject of proof. Call-time choice is essential, since bubbling is not correct with respect to run-time choice: in Example 1, *fdouble (g?h) 0* can be reduced with run-time choice to $0, 1$ or 2, while *fdouble g 0 ? fdouble h 0* leads only to 0 and 2.

The fact that bubbling preserves $HOCRWL_{let}$-semantics has a simple formulation:

Theorem 9 (Correctness of bubbling). *If* $e \to^{bub} e'$, *then* $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$. *In other terms,* $[\![\mathcal{C}[e_1?e_2]]\!] = [\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!]$ $(= [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!])$, *for any* $e_1, e_2 \in LExp$ *and context* $\mathcal{C}$.

From this and the equivalence results of Sect. 3 we obtain as immediate corollary the correctness of bubbling in terms of rewriting:

Corollary 1. $e \to_l^* t \Leftrightarrow e\ (\to_l \cup \to_{bub})^*\ t$

It is interesting to observe that most of the proof of Th. 9 consists of direct calculations with denotation of expressions, in the form of chains of equalities of denotations, justified by general properties of the semantics like Th. 1. We find this methodology quite appealing and for this reason we include (a part of) the proof.

Proof (For Theorem 9, Correctness of bubbling). The proof uses the following easy (not proved here) lemma about semantics of ?, which justifies also the equation $[\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!] = [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!]$ stated in the Theor. 9.

Lemma 7. $[\![e_1?e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$, *for any* $e_1, e_2 \in LExp_\bot$.

Now, we reason by induction on the number k of *let*'s occurring in $\mathcal{C}[e_1?e_2]$.

- $k = 0$: Since there is no *let* in $e_1?e_2$, we can apply Theor. 1 to obtain:

$$\begin{array}{ll}
[\![\mathcal{C}[e_1?e_2]]\!] & = \textit{(by Theor. 1)} \\
\bigcup_{t\in[\![e_1?e_2]\!]} [\![\mathcal{C}[t]]\!] & = \textit{(by Lemma 17)} \\
\bigcup_{t\in([\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!])} [\![\mathcal{C}[t]]\!] & = \textit{(set operations)} \\
\bigcup_{t\in[\![\mathcal{C}[e_1]]\!]} [\![\mathcal{C}[t]]\!] \ \cup\ \bigcup_{t\in[\![\mathcal{C}[e_2]]\!]} [\![\mathcal{C}[t]]\!] & = \textit{(by Theor. 1)} \\
[\![\mathcal{C}[e_1]]\!] \ \cup\ [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 17)} \\
[\![\mathcal{C}[e_1]\ ?\ \mathcal{C}[e_2]]\!] &
\end{array}$$

- $k > 0$: We reason by induction on the structure of $\mathcal{C}$. The most interesting case is that of let bindings:

– $\mathcal{C} \equiv \textit{let } x = e \textit{ in } \mathcal{C}'$: then

$$\begin{array}{ll}
[\![\mathcal{C}[e_1?e_2]]\!] & = \\
[\![\textit{let } x{=}e \textit{ in } \mathcal{C}'[e_1?e_2]]\!] & = \textit{(by Theor. 2,}\sigma \equiv \{x/t\}\textit{)} \\
\bigcup_{t\in[\![e]\!]} [\![\mathcal{C}'[e_1?e_2]\sigma]\!] & = \\
\bigcup_{t\in[\![e]\!]} [\![\mathcal{C}'\sigma[e_1\sigma?e_2\sigma]]\!] & = \textit{(by IH on k, that decreases)} \\
\bigcup_{t\in[\![e]\!]} [\![\mathcal{C}'\sigma[e_1\sigma]?\mathcal{C}'\sigma[e_2\sigma]]\!] & = \textit{(by Lemma 17)} \\
\bigcup_{t\in[\![e]\!]} ([\![\mathcal{C}'\sigma[e_1\sigma]]\!] \ \cup\ [\![\mathcal{C}'\sigma[e_2\sigma]]\!]) & = \textit{(set operations)} \\
\bigcup_{t\in[\![e]\!]} [\![\mathcal{C}'\sigma[e_1\sigma]]\!] \ \cup\ \bigcup_{t\in[\![e]\!]} [\![\mathcal{C}'\sigma[e_2\sigma]]\!] & = \textit{(by Theor. 2)} \\
[\![\textit{let } x{=}e \textit{ in } \mathcal{C}'[e_1]]\!] \ \cup\ [\![\textit{let } x{=}e \textit{ in } \mathcal{C}'[e_2]]\!] & = \\
[\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 17)} \\
[\![\mathcal{C}[e_1]\ ?\ \mathcal{C}[e_2]]\!] &
\end{array}$$

6 Translation to first order

Since [28], a common technique to implement HO features in FO settings consists in a *HO-to-FO* translation introducing data constructors to represent partial applications and a special function @ (read *apply*) for reducing application of such constructors. This has been used within the context of *FLP* in [9, 4]. Here we adapt such a transformation to our context and provide a correctness proof with

respect to the semantics of the source and object programs, given by *HOCRWL* and *CRWL* [11, 19] respectively.

Definition 2 (First order translation). *Given a HOCRWL-program* $\mathcal{P} = \{f\ \overline{p_1} \to e_1, \ldots, f\ \overline{p_m} \to e_m\}$ *built up over the signature* $\Sigma = FS \cup CS$*, its* first order translation P_{fo} *will be defined over the* **extended signature** $\Sigma_{fo} = FS_{fo} \cup CS_{fo}$ *where:*

$$FS_{fo} = FS \cup \{@\}; \quad CS_{fo} = \bigcup_{c \in CS^n, n \in \mathbb{N}} \{c_0, \ldots, c_n\} \cup \bigcup_{f \in FS^n, n \in \mathbb{N}} \{f_0, \ldots, f_{n-1}\}$$

being @ *a new function symbol of arity 2 and* $c_0, \ldots, c_n, f_0, \ldots, f_{n-1}$ *new symbols (with arities indicated by the sub-index). The set* $\mathcal{P}_@$ *of* $@-rules$ *is defined as:*

$$\begin{array}{ll} @(c_k(X_1, \ldots, X_k), Y) & = c_{k+1}(X_1, \ldots, X_k, Y), \text{for each } c \in DC^n, k < n \\ @(f_k(X_1, \ldots, X_k), Y) & = f_{k+1}(X_1, \ldots, X_k, Y), \text{for each } f \in FS^n, k+1 < n \\ @(f_{n-1}(X_1, \ldots, X_{n-1}), Y) & = f(X_1, \ldots, X_{n-1}, Y), \text{for each } f \in FS^n \end{array}$$

The transforming function $fo : Exp_{\Sigma,\bot} \to Exp_{\Sigma_{fo},\bot}$ *is defined as:*

$fo(\bot) = \bot$ $\quad fo(X) = X$ $\quad fo(h) = h_0$, *if* $h \in CS$ *or* $h \in FS^n, n > 0$
$fo(f) = f$, *if* $f \in FS^0$ $\quad fo(e_1\ e_2) = @(fo(e_1), fo(e_2))$

The **transformed program** *is defined as* $P_{fo} = \{f(\overline{fo(p_1)\downarrow_@}) \to fo(e_1)\downarrow_@, \ldots, f(\overline{fo(p_m)\downarrow_@}) \to fo(e_m)\downarrow_@\} \cup P_@$*, where* $e\downarrow_@$ *stands for a* **normal form** *for* e *with respect to* $@-rules$ *defined above.*

The program rules obtained by the transformation are well defined: it is easy to prove that if p is a pattern then $fo(p)\downarrow_@$ is a FO constructor term.

For the program of Example 1 we have $FS_{fo} = \{+, f, g, h, f', fadd, fdouble, @\}$ and $CS_{fo} = \{0, s_0, s, +_0, +_1, g_0, h_0, {f'}_0, fadd_0, fadd_1, fadd_2, fdouble_0\}$. The translated rules are:

$$g(X) \to 0 \quad f \to g_0 \quad f \to h_0 \quad f'(X) \to @(f, X) \quad h(X) \to s(0)$$
$$fadd(F, G, X) \to @(F, X) + @(G, X) \quad fdouble(F) \to fadd_2(F, F)$$

And the rules for @ are:

$$\begin{array}{lll} @(+_0, X) \to +_1(X) & @(s_0, X) \to s(X) & @(h_0, X) \to h(X) \\ @(+_1(X), Y) \to X + Y & @(g_0, X) \to g(X) & @(f'_0, X) \to f'(X) \\ @(fadd_0, F) \to fadd_1(F) & @(fadd_2(F, G), X) \to fadd(F, G, X) & \\ @(fadd_1(F), G) \to fadd_2(F, G) & @(fdouble_0, F) \to fdouble(F) & \end{array}$$

The translation of the expressions to reduce in that example are:
$fo(fdouble\ f\ 0)\downarrow_@ = @(fdouble(f), 0)$ $\quad fo(fdouble\ f'\ 0)\downarrow_@ = @(fdouble(f'_0), 0)$

In general we cannot expect to prove a statement of the form $fo(e) \twoheadrightarrow fo(t)$ because $fo(t)$ can contain calls to the function @, i.e. $fo(t)$ might not be a FO constructor term. But the same statement makes sense in the form $fo(e) \twoheadrightarrow fo(t)\downarrow_@$ because $fo(t)\downarrow_@$ is a FO constructor term.

Proposition 2. $[\![fo(e)\downarrow_@]\!]^{\mathcal{P}}_{CRWL} = [\![fo(e)]\!]^{\mathcal{P}}_{CRWL}$*. Moreover* $[\![fo(e)]\!] = [\![e']\!]$ *where* e' *is any expression obtained from* e *by reducing some calls of* @*.*

According to this, when proving a statement $fo(e) \twoheadrightarrow t$ we can use any equivalent expression e' (in the sense of previous lemma) in the left hand side and prove $e' \twoheadrightarrow t$.

The correctness of the transformation can be stated then as follows:

Theorem 10 (Adequacy of HO-to-FO translation). *Let $\mathcal{P}$ be a program, $e \in Exp_\perp$, $t \in Pat_\perp$. Then:* $\mathcal{P} \vdash_{HOCRWL} e \twoheadrightarrow t \Leftrightarrow \mathcal{P}_{fo} \vdash_{CRWL} fo(e) \twoheadrightarrow fo(t)\downarrow_@$ *Or, in terms of* HOlet*-rewriting:* $e \rightarrow^{!*} t \Leftrightarrow fo(e) \rightarrow^{!*} fo(t)\downarrow_@$

7 Conclusions

Our paper addresses the broad question: *what means 'reduction' for functional logic programming?*, which had no previous satisfactory answer for the combination *HO + non-deterministic functions + call-time choice* supported by current systems in the mainstream of the field (Curry [16], Toy [20]). This leads to subtle behaviors well characterized from the point of view of a declarative semantics [7], but with no corresponding basic notion of one-step reduction. We have made a number of identifiable **contributions** in this sense:

• We propose a notion of rewriting with local bindings (*HOlet-rewriting*) suitable for a large class of HO systems (possibly non-confluent and non-terminating, allowing extra variables in right-hand sides and HO-patterns in left-hand sides).

• We have proved equivalence of *HOlet*-rewriting wrt to *HOCRWL* [7] declarative semantics. Along the way we have extended *HOCRWL* to cope with *let*s, and established new compositional properties of *HOCRWL* semantics.

• We have lifted *HOlet*-rewriting to a notion of *HOlet-narrowing* which is able to bind variables to patterns, even HO ones representing intensional descriptions of functions. We prove soundness and completeness of *HOlet*-narrowing wrt. *HOlet*-rewriting.

• We have recast within our framework the definition and proof of correctness of *bubbling*, an operational rule investigated in [3, 2] using term graph rewriting techniques. Apart from extending it to HO, this case study illustrates quite well the power of using indistinctly rewriting and/or semantic-based reasoning.

• To close the panorama, we have formally proved that *translation from HO to FO*, a technique actually used in the implementations of FLP systems, still works properly when *let*-bindings with call-time choice are considered, while previous works [9, 4] consider only deterministic functions.

The first three points have been conceived as an extension to HO of our previous work on the FO case [19, 18]. However, adapting it has not been routine; on the contrary, some results have been indeed a technical challenge.

Our wish with this work, jointly with [19, 18], is to have provided foundational pieces useful to understand how a FLP computation proceeds, serving also as suitable technical basis to address in the call-time choice context other operational issues (rewriting and narrowing strategies, residuation, program optimization, types in computations,...), all of which are lines of future work.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *J. of Symb. Comp.*, 40(1):795–829, 2005.
2. S. Antoy, D. Brown, and S. Chiang. On the correctness of bubbling. *Proc. RTA'06)*, 35–49. Springer LNCS 4098, 2006.
3. S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. Termgraph'06,*, 61–70, ENTCS 176(1), 2007.
4. S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. *Proc. FLOPS'99*, 335–353, Springer LNCS 1722, 1999.
5. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Proc. POPL'95*, 233–246, 1995.
6. R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. *Proc. JICSLP'98*, 325 – 340. MIT Press, 1998.
7. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. *Proc. ICLP'97*, 153–167. MIT Press, 1997.
8. J. González-Moreno, T. Hortalá-González, and Rodríguez-Artalejo, M. Polymorphic types in functional logic programming. *J. of Functional and Logic Programming*, volume 2001/S01, 1–71, 2001.
9. J. C. González-Moreno. A correctness proof for warren's ho into fo translation. *Proc. GULP'93*, 569–584, 1993.
10. J. C. González-Moreno, M. T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. *Proc. CSL'92*, 216–230, Springer LNCS 702, 1992.
11. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. of Logic Programming*, 40(1):47–87, 1999.
12. M. Hanus. The integration of functions into logic programming: From theory to practice. *J. of Logic Programming*, 19&20:583–628, 1994.
13. M. Hanus. Curry mailing list. http://www.informatik.uni-kiel.de/ curry/listarchive/0497.html, March 2007.
14. M. Hanus. Multi-paradigm declarative languages. *Proc. ICLP 2007*, 45–75. Springer LNCS 4670, 2007.
15. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *J. of Functional Programming*, 9(1):33–75, 1999.
16. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
17. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
18. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Narrowing for non-determinism with call-time choice semantics. *Proc. WLP'07*, 2007.
19. F. López-Fraguas, J. Rodríguez-Hortalá, J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. *Proc. PPDP'07*, 197–208. ACM, 2007.
20. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. *Proc. RTA'99*, 244–247. Springer LNCS 1631, 1999.
21. J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.
22. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

23. K. Nakahara, A. Middeldorp, and T. Ida. A complete narrowing calculus for higher-order functional logic programming. *Proc. PLILP'95*, 97–114, LNCS 882, 1995.
24. S.L. Peyton Jones (ed.). Haskell 98 Language and Libraries. The Revised Report. Cambridge Univ. Press, 2003.
25. D. Plump. Essentials of term graph rewriting. *ENTCS* 51, 2001.
26. F. van Raamsdonk. Higher-order rewriting. In *Term Rewriting Systems*, Cambridge Univ. Press, 2003.
27. M. Rodríguez-Artalejo. Functional and constraint logic programming. *Revised Lect. of Int. Summer School CCL'99*, 202–270. Springer LNCS 2002, 2001.
28. D. H. Warren. Higher-order extensions to prolog: are they needed? *Machine Intelligence 10*, 441–454. Ellis Horwood Ltd., 1982.

A Proofs of the results

We give here the proofs of the results. The results themshelves have not been repeated. The appendix includes also many more auxiliary results. For the sake of readability we have sectioned the appendix following the structure of the paper. Many times we refer to proofs 'for the first order case'. This points to proofs in [19] or [18] (long version in `http://gpd.sip.ucm.es/fraguas/papers/longWLP07.pdf`).

2 Preliminaries: HOCRWL

Proof (For Theorem 1, Compositionality of HOCRWL semantics). We prove both implications in (i), and for the $\Rightarrow$ part of (i) we prove also (ii).
$\boxed{\Rightarrow}$ Assume $\mathcal{C}[e] \twoheadrightarrow t$.
The cases when $t = \bot$ or $\mathcal{C} = [\,]$ are trivial just taking $s = t$ (and in these cases (ii) does not apply).

For the rest of the cases, we reason by complete induction on the size K of the derivation of $\mathcal{C}[e] \twoheadrightarrow t$. We assume then that the implication is true for any size $< K$. We need in addition distinguish cases according to the shape of $\mathcal{C}[e] \twoheadrightarrow t$ and its derivation, following the rules of HOCRWL:

(RR) $\mathcal{C}[e] \twoheadrightarrow t \equiv X \twoheadrightarrow X$: this can only happen if $e = X$ and $\mathcal{C} = [\,]$

(DC) $\mathcal{C}[e] \twoheadrightarrow t \equiv h\ e_1 \dots e_n \twoheadrightarrow h\ t_1 \dots t_n$, where $h\ t_1 \dots t_n$ is a partial pattern. The derivation must take the form

$$\frac{e_1 \twoheadrightarrow t_1 \dots e_n \twoheadrightarrow t_n}{h\ e_1 \dots e_n \twoheadrightarrow h\ t_1 \dots t_n}$$

Now we split the proof in two subcases:

- $e = h\ e_1 \dots e_k$, for $k < n$ ($k = n$ corresponds to $\mathcal{C} = [\,]$). We take $s = h\ t_1 \dots t_k$, for which there are derivations $e \twoheadrightarrow s$ and $\mathcal{C}[s] \twoheadrightarrow t$ given by
$$\frac{e_1 \twoheadrightarrow t_1 \dots e_k \twoheadrightarrow t_k}{h\ e_1 \dots e_k \twoheadrightarrow h\ t_1 \dots t_k}$$
and
$$\frac{t_1 \twoheadrightarrow t_1 \dots t_k \twoheadrightarrow t_k\ e_{k+1} \twoheadrightarrow t_{k+1} \dots e_n \twoheadrightarrow t_n}{h\ t_1 \dots t_k\ e_{k+1} \dots e_n \twoheadrightarrow h\ t_1 \dots t_n}$$
with sizes $< K$ and $\leq K$ respectively[4] .
- $e_i = \mathcal{C}'[e]$, for some $1 \leq i \leq n$. Then we have $\mathcal{C}'[e] \twoheadrightarrow t_i$ with size $K' < K$. If $t_i = \bot$ or $\mathcal{C}' = [\,]$, it is enough to take $s = t_i$. Otherwise, by the HI, there exists s such that $e \twoheadrightarrow s$ with derivation of size $< K'$ (and hence $< K$) and $\mathcal{C}'[s] \twoheadrightarrow t_i$ with derivation of size $\leq K'$. Since $\mathcal{C}[s] \equiv h\ e_1 \dots \mathcal{C}'[s] \dots e_n$, we can build the following derivation of $\mathcal{C}[s] \twoheadrightarrow t$

[4] We are using here (and we will do several more times) the easy fact that for any pattern t, $t \twoheadrightarrow t$ can be proved with a derivation not larger than any other derivation $e \twoheadrightarrow t$.

$$\frac{e_1 \twoheadrightarrow t_1 \ldots \mathcal{C}'[s] \twoheadrightarrow t_i \ldots e_n \twoheadrightarrow t_n}{h\ e_1 \ldots \mathcal{C}'[s] \ldots e_n \twoheadrightarrow h\ t_1 \ldots t_i \ldots t_n}$$

of size $\leq K$.

(OR) $\mathcal{C}[e] \twoheadrightarrow t \equiv f\ e_1 \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t$ where $f \in FS^n$. The derivation must have the form

$$\frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n \quad r\ a_1 \ldots a_m \twoheadrightarrow t}{f\ e_1 \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t}$$

where $f\ t_1 \ldots t_n \to r\ \in [\mathcal{P}]_\perp$.

Now we split the proof in four subcases:

- $e = f\ e_1 \ldots e_k$, for $k < n$. We take $s = f\ t_1 \ldots t_k$, for which there are derivations $e \twoheadrightarrow s$ and $\mathcal{C}[s] \twoheadrightarrow t$ given by
$$\frac{e_1 \twoheadrightarrow t_1 \ldots e_k \twoheadrightarrow t_k}{f\ e_1 \ldots e_k \twoheadrightarrow h\ t_1 \ldots t_k}$$
and
$$\frac{t_1 \twoheadrightarrow t_1 \ldots t_k \twoheadrightarrow t_k\ e_{k+1} \twoheadrightarrow t_{k+1} \ldots e_n \quad r\ a_1 \ldots a_m \twoheadrightarrow t_n}{f\ t_1 \ldots t_k\ e_{k+1} \ldots e_n\ a_1 \ldots a_m \twoheadrightarrow t}$$
with sizes $< K$ and $\leq K$ respectively.
- $e = f\ e_1 \ldots e_n\ a_1 \ldots a_k$, for $0 \leq k < m$, which corresponds to $\mathcal{C} = [\ ]\ a_{k+1} \ldots a_m$ ($k = m$ would correspond to $\mathcal{C} = [\ ]$). Now, consider $e' = r\ a_1 \ldots a_k$, so that $r\ a_1 \ldots a_m = \mathcal{C}[e']$. Since $r\ a_1 \ldots a_m \twoheadrightarrow t$ has a derivation of size $K' < K$, we can apply the IH to $\mathcal{C}[e']$ (since $t \neq \perp, \mathcal{C} \neq [\ ]$) obtaining s and derivations for $e' \twoheadrightarrow s$ and $\mathcal{C}[s] \twoheadrightarrow t$ with sizes $< K'$ and $\leq K'$. We only need to show that $e \twoheadrightarrow s$ for which we can build the derivation
$$\frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n \quad r\ a_1 \ldots a_k \twoheadrightarrow s}{f\ e_1 \ldots e_n\ a_1 \ldots a_k \twoheadrightarrow s}$$
which has size $< K$.
- $e_i = \mathcal{C}'[e]$, for some $1 \leq i \leq n$. In this case the proof proceed in a very similar way to the second case of (DC).
- $a_i = \mathcal{C}'[e]$, for some $1 \leq i \leq m$. Consider $\mathcal{C}'' = r\ a_1 \ldots \mathcal{C}' \ldots a_m$, so that $\mathcal{C}''[e] = r\ a_1 \ldots a_i \ldots a_m$, and therefore we have a derivation of $\mathcal{C}''[e] \twoheadrightarrow t$ of size $K' < K$. By the IH, there exist s and derivations for $e \twoheadrightarrow s$ and $\mathcal{C}''[s] \twoheadrightarrow t$ os sizes $< K'$ (hence $< K$) and $\leq K'$. Now, just notice that $\mathcal{C}''[s] = r\ a_1 \ldots s \ldots a_m$ and $\mathcal{C}[s] = f\ e_1 \ldots e_n\ a_1 \ldots s \ldots a_m$, and therefore we have a derivation for $\mathcal{C}[s] \twoheadrightarrow t$ of the form
$$\frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n \quad r\ a_1 \ldots s \ldots a_m \twoheadrightarrow t}{f\ e_1 \ldots e_n\ a_1 \ldots s \ldots a_m \twoheadrightarrow t}$$
and with size $\leq K$.

$\boxed{\Leftarrow}$ The proof becomes almost immediate if we use the calculus HOBRC of [7], which is shown there to be equivalent to the calculus of Fig. 1 for proving statements $e \twoheadrightarrow t$. Notice however that HOBRC is able to prove also more general statements of the form $e \twoheadrightarrow e'$, for which the following simple stability can be proved by a straightforward by induction on the structure of $\mathcal{C}$:

Lemma 8. *$e \twoheadrightarrow e' \ \Rightarrow\ \mathcal{C}[e] \twoheadrightarrow \mathcal{C}[e']$, for any e, e'*

We can now proceed with the proof of $\Leftarrow$. Assume $e \twoheadrightarrow s$ and $\mathcal{C}[s] \twoheadrightarrow t$. Then, by the previous lemma $\mathcal{C}[e] \twoheadrightarrow \mathcal{C}[s]$, and by the transitivity rule of HOBRC we obtain $\mathcal{C}[e] \twoheadrightarrow t$ as desired. Notice that transitivity of $\twoheadrightarrow$ is the only rule used from HOBRC.

3 Higher order *let*-rewriting

Firstly we introduce an alternative (but equivalent) version of the $HOCRWL_{let}$ calculus that will make easier some proofs in this section and also in Sect. 6. The variant $HOCRWL'_{let}$ is presented in Figure 4.

(B) $\dfrac{}{e \twoheadrightarrow \bot}$ **(RR)** $\dfrac{}{x \twoheadrightarrow x}$ $x \in \mathcal{V}$

(DC) $\dfrac{e_1 \twoheadrightarrow t_1 \ \ldots \ e_m \twoheadrightarrow t_m}{h\ e_1 \ldots e_m \twoheadrightarrow h\ t_1 \ldots t_m}$ $h \in \Sigma$, if $h\ t_1 \ldots t_m$ is a partial pattern, $m \geq 0$

(OR) $\dfrac{e_1 \twoheadrightarrow t_1 \ldots \ e_n \twoheadrightarrow t_n \quad r \twoheadrightarrow t}{f\ e_1 \ldots e_n \twoheadrightarrow t}$ if t is a partial pattern, $(f\ t_1, \ldots, t_n \to r) \in [\mathcal{P}]_\bot$

(Let) $\dfrac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t}$ if t is a partial pattern

(Ap) $\dfrac{e_1 \twoheadrightarrow t_1 \quad e_2 \twoheadrightarrow t_2 \quad t_1\ t_2 \twoheadrightarrow t}{e_1\ e_2 \twoheadrightarrow t}$

Fig. 4. $HOCRWL'_{let}$: an alternative to $HOCRWL_{let}$

With respect to the original version $HOCRWL_{let}$ introduced in Sect. 3, in this calculus the rules **(OR)** and **(Let)** are modified and a new rule **(Ap)** is added. Both presentations are equivalent in the sense that they prove exactly the same approximation statements as shows the following result.

Lemma 9 (Equivalence of $HOCRWL_{let}$ and $HOCRWL'_{let}$). *For any let-expression e we have $[\![e]\!]^{HOCRWL_{let}} = [\![e]\!]^{HOCRWL'_{let}}$.*

Proof. It is a direct application of Lemma 2. The only case in which the derivations are really different is when $e = (e_1\ e_2)$. In this case by the cited Lemma we have in $HOCRWL_{let}$:

$$t \in [\![e_1\ e_2]\!] \Leftrightarrow t \in \bigcup_{t_1 \in [\![e_1]\!]} \bigcup_{t_2 \in [\![e_2]\!]} [\![t_1\ t_2]\!] \Leftrightarrow \exists t_1 \in [\![e_1]\!], t_2 \in [\![e_2]\!].t_1\ t_2 \twoheadrightarrow t$$

But then, in $HOCRWL'_{let}$ we can build the proof:

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2 \twoheadrightarrow t_2 \quad (t_1\ t_2) \twoheadrightarrow t}{(e_1\ e_2) \twoheadrightarrow t}\ AP$$

Notice that the implications are bidirectional.

Proof (For Theorem 2, Weak compositionality of HOCRWL$_{let}$ *semantics).*

The proof becomes easier using an alternative version of *HOCRWL'*$_{let}$ introduced in Fig 4.

We must prove $C\ [e] \twoheadrightarrow t \Leftrightarrow \exists s$ such that $e \twoheadrightarrow s$ and $C\ [s] \twoheadrightarrow t$. By induction on the size of the proof for $C\ [e] \twoheadrightarrow t$. The base case only allows the proofs $C\ [e] \to \bot$, $C\ [e] \equiv X \to X$ and $C\ [e] \equiv h \to h$ with $h \in Pat$, that are clear.

Now, for the inductive step we consider the rule applied:

(DC) If the proof is:

$$\frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{h\ e_1\ \ldots\ e_n \twoheadrightarrow h\ t_1\ \ldots\ t_n}$$

and $C\ [e] = h\ e_1\ \ldots\ e_n$ we have two possible situations:

- $C\ [e] = h\ e_1\ \ldots (C'\ [e]) \ldots e_n$, i.e., e is a subexpression of some e_i. In this case, by i.h. $C'\ [e] \twoheadrightarrow t_i' \Leftrightarrow \exists s \in [\![e]\!]$ such that $C'\ [s] \twoheadrightarrow t_i'$. Then $C\ [s] = h\ e_1 \ldots (C'\ [s]) \ldots e_n$ and it is direct to build the proof for $C\ [s] \twoheadrightarrow t$.
- $C\ [e] = C'\ [h\ e_1 \ldots e_k]$ with $C' = []\ e_{k+1} \ldots e_n$ (with $k \geq 0$). We can take $s = h\ t_1 \ldots t_k \in [\![e]\!]$ and build the proof:

$$\frac{h\ t_1\ \ldots t_k \twoheadrightarrow h\ t_1\ \ldots t_k \quad e_{k+1} \twoheadrightarrow t_{k+1} \ldots e_n \twoheadrightarrow t_n}{C\ [s] = (h\ t_1\ \ldots t_k)\ e_{k+1}\ \ldots\ e_n \twoheadrightarrow h\ t_1\ \ldots t_k\ t_{k+1} \ldots t_n}$$

(OR) The proof is similar to the previous case.

(Let) It the proof is:

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t}$$

We have two possible situations:

- $C\ [e] = (let\ X = C'\ in\ e_2)\ [e]$, with $e_1 = C'\ [e]$ (e is a subexpression of e_1). By i.h. we have $e_1 \twoheadrightarrow t_1 \Leftrightarrow \exists s \in [\![e]\!]$ such that $C'\ [s] \twoheadrightarrow t_1$. Then we have:

$$\frac{C'\ [s] \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t_2}{C\ [s] \equiv let\ X = C'\ [s]\ in\ e_2 \twoheadrightarrow t}$$

- $C\ [e] = (let\ X = e_1\ in\ C')\ [e]$, with $e_2 = C'\ [e]$ (e is a subexpression of e_2). We have $e_2[X/t_1] = (C'\ [e])[X/t_1] = C'[X/t_1]\ [e[X/t_1]]$, but as $X \in BV(C)$ it must be $X \notin FV(e)$, and then $e_2[X/t_1] = C'[X/t_1]\ [e]$. By i.h. $e_2[X/t_1] \equiv C'[X/t_1]\ [e] \twoheadrightarrow t_2 \Leftrightarrow \exists s \in [\![e]\!]$ such that $C'[X/t_1]\ [s] \twoheadrightarrow t$. Now we can build:

$$\frac{e_1 \twoheadrightarrow t_1 \quad (C'\ [s])[X/t_1] \equiv^{(*)} C'[X/t_1]\ [s] \twoheadrightarrow t}{C\ [s] \equiv let\ X = e_1\ in\ C'\ [s] \twoheadrightarrow t}$$

The equivalence (*) comes from: as $X \notin FV(e)$ and $e \to s$ then $X \notin var(s)$ (s can introduce fresh variables, but not X).

(Ap) It the proof is:

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2 \twoheadrightarrow t_2 \quad t_1\ t_2 \twoheadrightarrow t}{e_1\ e_2 \twoheadrightarrow t}$$

Again we have two possible situations:

- $C\ [e] = (C'\ e_2)\ [e]$ being e a subexpression of e_1. By i.h. we have $C'\ [e] \twoheadrightarrow t_1 \Leftrightarrow \exists s \in [\![e]\!]$ such that $C'\ [s] \twoheadrightarrow t_1$. It is easy to build the proof for $(C'\ e_2)\ [s] \twoheadrightarrow t$
- $C\ [e] = (e_1\ C')\ [e]$ being e a subexpression of e_2. Similar.

This ends the proof of the main part of the theorem. With respect to consequences *(i)*, *(ii)* , and *(iii)*, the first two follow easily: for *(i)*, $(e\ e') \equiv \mathcal{C}[[]e]$ where $\mathcal{C} = ([\ ]\ e')$, and we can rename bound variables in e' to avoid clashes with free variables of e. Then it suffices to apply the main part. The case *(ii)* is similar.

For (iii), we can reason again that $let\ x = e\ in\ e' \equiv \mathcal{C}[[]e]$ where $\mathcal{C} = let\ x = [\ ]\ in\ e'$, and from the main part (as before, renaming in e' if necessary) we obtain $[\![let\ x = e\ in\ e']\!] = \bigcup_{t \in [\![e]\!]} [\![let\ x = t\ in\ e']\!]$. Now, using that $let\ x = t\ in\ e'$ is not inside any context and $t \in Pat_\bot$, it is easy to prove, by reasoning directly over the HOCRWL rule for *(Let)*, that $[\![let\ x = t\ in\ e']\!] = [\![e'\sigma]\!]$, where $\sigma = \{X/t\}$.

Proof (For Proposition 1). We define for any $e \in LExp$ the size (k_1, k_2, k_3, k_4), where

$k_1 \equiv$ *number of subexpressions in* e *to which (LetAp) is applicable.*
$k_2 \equiv$ *number of subexpressions in* e *to which (LetIn) is applicable.*
$k_3 \equiv$ *number of* lets *in* e.
$k_4 \equiv$ *sum of the levels of nesting of all* let*-subexpressions in* e.

Sizes are lexicographically ordered. We prove now that application of *(LetIn), (Bind), (Elim), (Flat), (LetAp)* in any context (hence, also the application of *(Contxt)*) decreases the size, what proves termination of $\rightarrow^l \backslash FApp$. The effect of each rule in the size is summarized as follows (in each case, we stop at the decreasing component):

(LetIn): $(=, <, \_, \_)$
(Bind): $(=, =, <, \_)$
(Elim): $(\leq, \leq, <, \_)$
(Flat): $(=, =, =, <)$
(LetAp): $(<, \_, \_, \_)$

Lemma 10 (Substitution lemma for *let*-expressions). *Given* $e, e' \in LExp_\bot$, $\theta \in LSubst_\bot$ *and* $X \in \mathcal{V}$ *such that* $X \notin dom(\theta)$ *and* $X \notin vRan(\theta)$, *then*

$$(e[X/e'])\theta \equiv e\theta[X/e'\theta]$$

Proof. This proof almost identical to the corresponding proof in first order, which proceeds by induction on the structure of e', just replacing the case for $e \equiv h(e_1, \ldots, e_n)$ with the following cases:

– $e \equiv h \in \Sigma$: trivial, because h is not affected by any substitution.

– $e \equiv e_1\ e_2$: Then

$$(e[X/e'])\theta \equiv (e_1\ e_2)[X/e']\theta \equiv (e_1[X/e']\theta)\ (e_2[X/e']\theta)$$
$$\equiv_{HI} (e_1\theta[X/e'\theta])\ (e_2\theta[X/e'\theta]) \equiv (e_1\ e_2)\theta[X/e'\theta] \equiv e\theta[X/e'\theta]$$

Lemma 11. *For any $\theta \in Susbst_\perp$, $\mathcal{C} \in Cntxt$ and $e \in LExp_\perp$ such that $dom(\theta) \cap BV(\mathcal{C}) = vRan(\theta) \cap BV(\mathcal{C}) = \emptyset$, we have $(\mathcal{C}[e])\theta \equiv \mathcal{C}\theta[e\theta]$.*

Proof. This proof is again very similar to the corresponding proof in first order, just replacing the case for first order application with the cases for the two kinds of higher order application context:

– $\mathcal{C} \equiv \mathcal{C}'\ a$: Then

$$(\mathcal{C}[e])\theta \equiv (\mathcal{C}'[e]\ a)\theta \equiv (\mathcal{C}'[e]\theta)\ (a\theta) \equiv_{IH} (\mathcal{C}'\theta[e\theta])\ (a\theta)$$
$$\equiv ((\mathcal{C}'\theta[])\ (a\theta))[e\theta] \equiv ((\mathcal{C}'[]\ a)\theta)[e\theta] \equiv \mathcal{C}\theta[e\theta]$$

– $\mathcal{C} \equiv a\ \mathcal{C}'$: Then

$$(\mathcal{C}[e])\theta \equiv (a\ (\mathcal{C}'[e]))\theta \equiv (a\theta)\ (\mathcal{C}'[e]\theta) \equiv_{IH} (a\theta)\ (\mathcal{C}'\theta[e\theta])$$
$$\equiv ((a\theta)\ (\mathcal{C}'\theta[]))[e\theta] \equiv ((a\ (\mathcal{C}'[]))\theta)[e\theta] \equiv \mathcal{C}\theta[e\theta]$$

Lemma 12. $\forall t \in Pat_\perp, |t| \equiv t$

Proof. A simple induction on the structure of patterns.

Lemma 13. *For any $h \in \Sigma, \sigma \in Susbt_\perp, e_1, \ldots, e_m \in LExp_\perp, m \geq 0$ we have $(h\ e_1 \ldots e_m)\sigma \equiv h\ (e_1\sigma) \ldots (e_m\sigma)$*

Proof. A simple induction over m, using left associativity of application.

Lemma 14. *For any $e \in LExp_\perp, t \in Pat_\perp$:*

a) $|let\ X = e\ in\ t| \equiv t[X/|e|]$
b) $|t[X/e]| \equiv t[X/|e|]$

Proof. a) is just a consequence of *b)*, as $|let\ X = e\ in\ t| \equiv |t[x/e]| \equiv_{b)} t[X/|e|]$. And we can prove *b)* by induction over the structure of the pattern t.

Base cases

– $t \equiv X$. $|t[X/e]| \equiv |X[X/e]| \equiv |e|| \equiv X[X/|e|] \equiv t[X/|e|]$

– $t \equiv Y \not\equiv X$. $|t[X/e]| \equiv |Y[X/e]| \equiv |Y| \equiv Y \equiv Y[X/|e|] \equiv t[X/|e|]$

– $t \equiv h \in CS^m, m \geq 0$ or $h \in FS^n, n > 0$. $|t[X/e]| \equiv |h[X/e]| \equiv |h| \equiv h \equiv h[X/|e|] \equiv t[X/|e|]$

Inductive step

– $t \equiv c\ t_1 \ldots t_m, c \in CS^n, 0 \leq m \leq n$. $|t[X/e]| \equiv |(c\ t_1 \ldots t_m)[X/e]|$
$\equiv_{lemma13} |c\,(t_1[X/e]) \ldots (t_m[X/e])| \equiv_{\text{def. of shell}} c\,|t_1[X/e]| \ldots |t_m[X/e]|$
$\equiv_{IH} c\ (t_1[X/|e|]) \ldots (t_m[X/|e|]) \equiv_{lemma13} (c\ t_1 \ldots t_m)[X/|e|] \equiv t[X/|e|]$

– $t \equiv f\ t_1 \ldots t_m, f \in FS^n, 0 \leq m < n$. Very similar to the previous case.

Proof (For Lemma 1, Monotonicity of hypersemantics). By induction on the structure of the context $\mathcal{C}$:

$$Cntxt \ni \mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$$

Base case :

– $\mathcal{C} \equiv []$: then $\mathcal{C}[e] \equiv e$ and $\mathcal{C}[e'] \equiv e'$, therefore the lemma follows from the hypothesis.

Inductive step :

– $\mathcal{C} \equiv \mathcal{C}'\ a$: so $\mathcal{C}[e] \equiv (\mathcal{C}'[e])\ a$. Let $\theta \in PSubst_\perp$ be such that $((\mathcal{C}'[e])\ a)\theta \equiv ((\mathcal{C}'[e])\theta)\ (a\theta) \twoheadrightarrow t$. Then by theorem 2 there must exist $s_1 \in [\![(\mathcal{C}'[e])\theta]\!]$ such that $s_1\ (a\theta) \twoheadrightarrow t$. But by IH $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$, so $s_1 \in [\![(\mathcal{C}'[e'])\theta]\!]$ and $((\mathcal{C}'[e'])\theta)\ (a\theta) \twoheadrightarrow t$ by theorem 2 again we have $(\mathcal{C}[e'])\theta \twoheadrightarrow t$.

– $\mathcal{C} \equiv a\ \mathcal{C}'$: so $\mathcal{C}[e] \equiv a\ (\mathcal{C}'[e])$. Let $\theta \in PSubst_\perp$ be such that $(a\ (\mathcal{C}'[e]))\theta \equiv (a\theta)\ ((\mathcal{C}'[e])\theta) \twoheadrightarrow t$. Then by theorem 2 there must exists $s_2 \in [\![(\mathcal{C}'[e])\theta]\!]$ such that $(a\theta)\ s_2 \twoheadrightarrow t$. But by IH $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$, so $s_2 \in [\![(\mathcal{C}'[e'])\theta]\!]$ and $(a\theta)\ ((\mathcal{C}'[e'])\theta) \twoheadrightarrow t$ by theorem 2 again we have $(\mathcal{C}[e'])\theta \twoheadrightarrow t$.

– $\mathcal{C} \equiv let\ X = \mathcal{C}'\ in\ e_1$: then $\mathcal{C}[e] \equiv let\ X = \mathcal{C}'[e]\ in\ e_1, \mathcal{C}[e'] \equiv let\ X = \mathcal{C}'[e']\ in\ e_1$. Let $\theta \in PSubst_\perp$ such that $(let\ X = \mathcal{C}'[e]\ in\ e_1)\theta \twoheadrightarrow t$, then the proof must be done using the rule **Let** in the form:

$$\frac{(\mathcal{C}'[e])\theta \twoheadrightarrow t_1\ \ e_1\theta[X/t_1] \twoheadrightarrow t}{let\ X = (\mathcal{C}'[e])\theta\ in\ e_1\theta \twoheadrightarrow t}\ Let$$

As $[\![e]\!] \Subset [\![e']\!]$, then by i.h. (because $\mathcal{C}'$ is a part of $\mathcal{C}$) we have $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$ and then $((\mathcal{C}'[e])\theta \twoheadrightarrow t_1)$ implies $((\mathcal{C}'[e'])\theta \twoheadrightarrow t_1)$. Then we have:

$$\frac{\dfrac{}{(\mathcal{C}'[e'])\theta \twoheadrightarrow t_1}\ IH\quad \dfrac{}{e_1\theta[X/t_1] \twoheadrightarrow t}\ Hyp}{let\ X = (\mathcal{C}'[e'])\theta\ in\ e_1\theta \twoheadrightarrow t}\ Let$$

– $\mathcal{C} \equiv let\ X = e_1\ in\ \mathcal{C}'$: then $\mathcal{C}[e] \equiv let\ X = e_1\ in\ \mathcal{C}'[e]$. Let $\theta \in PSubst_\perp$ such that $(let\ X = e_1\ in\ \mathcal{C}'[e])\theta \twoheadrightarrow t$, then the proof must be done by rule **Let** in the form:

$$\frac{e_1\theta \twoheadrightarrow t_1\ \ (\mathcal{C}'[e])\theta[X/t_1] \twoheadrightarrow t}{let\ X = e_1\theta\ in\ (\mathcal{C}'[e])\theta \twoheadrightarrow t}\ Let$$

We have $[\![e]\!] \Subset [\![e']\!]$ and then, by IH, $[\![\mathcal{C}'[e]]\!] \Subset [\![\mathcal{C}'[e']]\!]$. On the other hand $([X/t_1] \circ \theta) \in PSubst_\perp$, so $(\mathcal{C}'[e'])\theta[X/t_1] \twoheadrightarrow t$ and then:

$$\frac{\dfrac{}{e_1\theta \twoheadrightarrow t_1}\ Hyp\quad \dfrac{}{(\mathcal{C}'[e'])\theta[X/t_1] \twoheadrightarrow t}\ IH}{let\ X = e_1\theta\ in\ (\mathcal{C}'[e'])\theta \twoheadrightarrow t}\ Let$$

Proof (For Lemma 2, One-Step Hyper-Soundness of HOlet*-rewriting).* The cases of *(Contx), (Elim), (Bind), (Flat) and (Fapp)* are very similar to the first order case.

(LetAp) $(let\ X = e_1\ in\ e_2)e_3 \to^l let\ X = e_1\ in\ e_2e_3, \quad$ if $X \notin FV(e_3)$

Given $\theta \in PSusbt_\perp$ *such that* $\mathcal{P} \vdash_{CRWL_{let}} (let\ X = e_1\ in\ e_2e_3)\theta \twoheadrightarrow t$ *then* $\mathcal{P} \vdash_{CRWL_{let}} ((let\ X = e_1\ in\ e_2)e_3)\theta \twoheadrightarrow t$:
If $\mathcal{P} \vdash_{CRWL_{let}} (let\ X = e_1\ in\ e_2e_3)\theta \equiv let\ X = e_1\theta\ in\ (e_2\theta)(e_3\theta) \twoheadrightarrow t$, it must be with a proof:

$$\frac{e_1\theta \twoheadrightarrow t_1 \quad ((e_2\theta)(e_3\theta))[X/t_1] \twoheadrightarrow t}{let\ X = e_1\theta\ in\ (e_2\theta)(e_3\theta) \twoheadrightarrow t}\ Let$$

So we can build a proof for $\mathcal{P} \vdash_{CRWL_{let}} ((let\ X = e_1\ in\ e_2)e_3)\theta \equiv (let\ X = e_1\theta\ in\ e_2\theta)(e_3\theta) \twoheadrightarrow t$:

$$\frac{e_1\theta \twoheadrightarrow t_1 \quad (e_2\theta[X/t_1])(e_3\theta) \equiv ((e_2\theta)(e_3\theta))[X/t_1] \twoheadrightarrow t}{(let\ X = e_1\theta\ in\ e_2\theta)(e_3\theta) \twoheadrightarrow t}\ Let$$

Because, as $X \notin FV(e_3)$ by the premise of (LetAp) and $X \notin vRan(\theta)$ by the variable convention, then $X \notin FV(e_3\theta)$, so $e_3\theta \equiv e_3\theta[X/t_1]$ and also $(e_2\theta[X/t_1])(e_3\theta) \equiv (e_2\theta[X/t_1])(e_3\theta[X/t_1]) \equiv ((e_2\theta)(e_3\theta))[X/t_1]$

(LetIn) $e_1\ e_2 \to^l let\ X = e_2\ in\ e_1\ X$
if e_2 is an active expression, variable application, junk or $e_2 \equiv let\ Y = e'\ in\ e''$, with $X \in \mathcal{V}$ fresh.

Given $\theta \in PSusbt_\perp$ *such that* $\mathcal{P} \vdash_{CRWL_{let}} (let\ X = e_2\ in\ e_1\ X)\theta \twoheadrightarrow t$ *then* $\mathcal{P} \vdash_{CRWL_{let}} (e_1\ e_2)\theta \twoheadrightarrow t$:
If $\mathcal{P} \vdash_{CRWL_{let}} (let\ X = e_2\ in\ e_1\ X)\theta \twoheadrightarrow t$ it must be with a proof:

$$\frac{e_2\theta \twoheadrightarrow t_2 \quad ((e_1\theta)\ X)[X/t_2] \equiv (e_1\theta)\ t_2 \twoheadrightarrow t}{let\ X = e_2\theta\ in\ (e_1\theta)\ X \twoheadrightarrow t}\ Let$$

By the variable convention and the freshness of X, $(let\ X = e_2\ in\ e_1\ X)\theta \equiv let\ X = e_2\theta\ in\ (e_1\theta)\ X$ and $((e_1\theta)\ X)[X/t_2] \equiv (e_1\theta)\ t_2$. But then by theorem 2, as $t_2 \in [\![e_2\theta]\!]$ and $(e_1\theta)\ t_2 \twoheadrightarrow t$ then it must happen $(e_1e_2)\theta \equiv (e_1\theta)\ (e_2\theta) \twoheadrightarrow t$.

Proof (For Theorem 3, Soundness of HOlet*-rewriting). (i)* Lemma 2, plus an obvious induction over derivation lengths, implies that $[\![e_1]\!] \Subset [\![e_2]\!]$, and then $[\![e_1]\!] \subseteq [\![e_2]\!]$ (just take $\theta = \epsilon$). It can be shown that, for any e', $|e'| \in [\![e']\!]$. But then $|e'| \in [\![e]\!]$, what exactly means that $e \twoheadrightarrow |e'|$.
(ii) From *(i)*, we get $e \twoheadrightarrow |t|$. But $|t| = t$, since t is a pattern.

The following simple lemma will be useful to prove lemma 3:

Lemma 15. *For any* $h \in \Sigma, t_1, \ldots, t_m \in Pat$ *if* $h\ t_1 \ldots t_m \notin Pat$ *then we have* $|h\ t_1 \ldots t_m| = \perp$ *and* $h\ t_1 \ldots t_m$ *is an active expression or junk.*

Proof. By a case distinction:

- $h \in FS^n$: Then $m \geq n$ because otherwise $h\ t_1 \dots t_m \in Pat$, but then $|h\ t_1 \dots t_m| = \bot$ because it is an active expression.

- $h \in CS^n$: Then $m > n$ because otherwise $h\ t_1 \dots t_m \in Pat$, but then $|h\ t_1 \dots t_m| = \bot$ because it is junk.

Proof (For Lemma 3, Weak peeling lemma). Given $m \geq 0$ let $i : Exp \rightharpoonup \mathbb{N}$ [5] be defined as $i(h\ e_1 \dots e_m) = \min j \in \{1, \dots, m\} \mid e_j \in Exp \setminus Pat$ if there exists some $e_j \in Exp \setminus Pat$, $i(h\ e_1 \dots e_m) = m + 1$ otherwise, and $\pi : Exp \rightharpoonup \mathbb{N}$ be defined as $\pi(h\ e_1 \dots e_m) = m + 1 - i(h\ e_1 \dots e_m)$. Then it is easy to prove that for any expression $e \equiv h\ e_1 \dots e_m$ we have $0 \leq \pi(e) \leq m$. We proceed by induction over the lexicographic product $(size(e), \pi(e))$, where $size(e)$ is equal to the number of symbols of Σ or variables, appering in e.

Base cases $e \equiv h$: Then we are done with $h \rightarrow^{l^0} h$ for $\overline{X} = \emptyset$.

Inductive step $e \equiv h\ e_1 \dots e_m$ with $m > 0$. If $e_1 \dots e_m \in Pat$ then we are done with $h\ e_1 \dots e_m \rightarrow^{l^0} h\ e_1 \dots e_m$, for $\overline{X} = \emptyset$. Otherwise e has the shape $e \equiv h\ t_1 \dots t_{i-1}\ e_i \dots e_m$ with $i > 0$, $t_1, \dots, t_{i-1} \in Pat, e_i \in Exp \setminus Pat$. Then we can do a case distinction over e_i:

a) $e_i \equiv X\ \overline{e_i}$ with $\overline{e_i} \neq \emptyset$, because otherwise $e_i \equiv X \in Pat$. But then

$$\begin{aligned} & e \equiv ((h\ t_1 \dots t_{i-1})\ (X\ \overline{e_i}))\ e_{i+1} \dots e_m \\ & \rightarrow^{l} (let\ Y_i = X\ \overline{e_i}\ in\ h\ t_1 \dots t_{i-1}\ Y_i)\ e_{i+1} \dots e_m \quad \text{by (LetIn)} \\ & \rightarrow^{l^*} let\ Y_i = X\ \overline{e_i}\ in\ h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m \quad \text{by (LetAp*)} \end{aligned}$$

By (Rule*) we always mean zero o more applications of (Rule). Besides, $size(h\ t_1 \dots t_{i-1}\ Y_i \dots e_m) < size(e)$, as $X\ \overline{e_i}$, with $size(X\ \overline{e_i}) \geq 2$ (as $\overline{e_i} \neq \emptyset$), has been replaced by Y_i, with $size(Y_i) = 1$. So, by IH, $h\ t_1 \dots t_{i-1}\ Y_i \dots e_m \rightarrow^{l^*} let\ \overline{X = a}\ in\ h\ t_1 \dots t_m$ under the conditions stipulated, so

$$\begin{aligned} & let\ Y_i = X\ \overline{e_i}\ in\ h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m \\ & \rightarrow^{l^*} let\ Y_i = X\ \overline{e_i}\ in\ let\ \overline{X = a}\ in\ h\ t_1 \dots t_m \end{aligned}$$

Then $|X\ \overline{e_i}| = \bot$ as $\overline{e_i} \neq \emptyset$, and it is easy the check that all the conditions of the lemma are fulfilled.

[5] We use $\rightharpoonup$ to stress that this is a partial function, as it is only defined for functor applications.

b) $e_i \equiv h_i\ \overline{e_i} \notin Pat, h_i \in \Sigma$. As e properly contains e_i then $size(e_i) < size(e)$, so by IH $e_i \equiv h_i\ \overline{e_i} \to^{l^*} let\ \overline{X_i = a_i}\ in\ h_i\ \overline{t_i}$. But then:

$$\begin{array}{ll} e \equiv ((h\ t_1 \dots t_{i-1})\ (h_i\ \overline{e_i}))\ e_{i+1} \dots e_m & \\ \to^{l^*} ((h\ t_1 \dots t_{i-1})\ (let\ \overline{X_i = a_i}\ in\ h_i\ \overline{t_i}))\ e_{i+1} \dots e_m & \text{(IH)} \\ \to^{l^*} (let\ Y_i = (let\ \overline{X_i = a_i}\ in\ h_i\ \overline{t_i})\ in\ h\ t_1 \dots t_{i-1}\ Y_i)\ e_{i+1} \dots e_m & \text{(LetIn)} \\ \to^{l^*} let\ Y_i = (let\ \overline{X_i = a_i}\ in\ h_i\ \overline{t_i})\ in\ h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m & \text{(LetAp*)} \\ \to^{l^*} let\ \overline{X_i = a_i}\ in\ let\ Y_i = h_i\ \overline{t_i}\ in\ h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m & \text{(Flat*)} \end{array}$$

Besides, $size(h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m) \leq size(e)$, as e_i, with $size(e_i) \geq 1$, has been replaced by Y_i, with $size(Y_i) = 1$. But as $t_1, \dots, t_{i-1}, Y_i \in Pat$ then $\pi(h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m) \leq m+1-(i+1) = m-i < m+1-i = \pi(h\ t_1 \dots t_{i-1}\ e_i \dots e_m) = \pi(e)$, so by IH $h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m \to^{l^*} let\ \overline{X = a}\ in\ h\ t_1 \dots t_m$ under the conditions stipulated, so

$$\begin{array}{l} let\ \overline{X_i = a_i}\ in\ let\ Y_i = h_i\ \overline{t_i}\ in\ h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m \\ \to^{l^*} let\ \overline{X_i = a_i}\ in\ let\ Y_i = h_i\ \overline{t_i}\ in\ let\ \overline{X = a}\ in\ h\ t_1 \dots t_m \quad \text{by IH} \end{array}$$

And then we have two possibilities:

i) $h_i\ \overline{t_i} \in Pat$: Then

$$\begin{array}{l} let\ \overline{X_i = a_i}\ in\ let\ Y_i = h_i\ \overline{t_i}\ in\ let\ \overline{X = a}\ in\ h\ t_1 \dots t_m \\ \to^{l} let\ \overline{X_i = a_i}\ in\ (let\ \overline{X = a}\ in\ h\ t_1 \dots t_m)[Y_i / h_i\ \overline{t_i}] \qquad \text{by (Bind)} \end{array}$$

Note that by $t_i \equiv Y_i$ because $Y_i \in Pat$, as Y_i was fresh then it does not appear in $t_1, \dots, t_{i-1}, e_{i-1}, \dots, e_m$, so as $\overline{a}$ and $t_{i+1}, \dots, t_m$ come from the IH applied to $h\ t_1 \dots t_{i-1}\ Y_i\ e_{i+1} \dots e_m$ then Y_i cannot appear in $\overline{a}$ nor $t_{i+1}, \dots, t_m$. So

$$\begin{array}{l} let\ \overline{X_i = a_i}\ in\ (let\ \overline{X = a}\ in\ h\ t_1 \dots t_{i-1}\ Y_i\ t_{i+1} \dots t_m)[Y_i / h_i\ \overline{t_i}] \\ \equiv let\ \overline{X_i = a_i}\ in\ let\ \overline{X = a}\ in\ h\ t_1 \dots t_{i-1}\ (h_i\ \overline{t_i})\ t_{i+1} \dots t_m \end{array}$$

and it is easy the check that all the conditions of the lemma are fulfilled.

ii) $h_i\ \overline{t_i} \notin Pat$: Then by lemma 15 we have $|h_i\ \overline{t_i}| = \bot$, so it is easy the check that all the conditions of the lemma are fulfilled.

Proof (For Lemma 4). The consequence $t \sqsubseteq t'[\overline{X / \bot}]$ holds just applying lemma 14. For the rest of the lemma we proceed by induction on the size K of the proof for $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, measured as the number of rules applied.

Base cases The reasoning is identical to the FO case.

Inductive step Let us see which rule was applied in the root of the proof for $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$:

DC Then we have $e \equiv h\ e_1 \dots e_m$ with $m > 0$, $h \in CS^n$ with $m \leq n$ or $h \in FS^n$ with $m < n$, and the following proof:

$$\frac{e_1 \twoheadrightarrow t_1 \quad \dots \quad e_m \twoheadrightarrow t_m}{\underbrace{h\ e_1 \dots e_m}_{e} \twoheadrightarrow \underbrace{h\ t_1 \dots t_m}_{t}}\ DC$$

But then appliing Th. 1 over $h\ e_1 \dots e_m \twoheadrightarrow h\ t_1 \dots t_m$ we get:

- $h\ e_1 \dots e_{m-1} \twoheadrightarrow s_1$ for some $s_1 \in Pat_\perp$ and with $size(h\ e_1 \dots e_{m-1} \twoheadrightarrow s_1) < K$.
- $s_1\ e_m \twoheadrightarrow h\ t_1 \dots t_m$ with $size(s_1\ e_m \twoheadrightarrow h\ t_1 \dots t_m) \leq K$.

As $s_1\ e_m \twoheadrightarrow h\ t_1 \dots t_m \not\equiv \perp$ then $s_1 \not\equiv \perp$. But then, as by the conditions of **DC** it must happen $h \in CS^n$ with $m \leq n$ or $h \in FS^n$ with $m < n$, the only rule that could have been applied at the root of $h\ e_1 \dots e_{m-1} \twoheadrightarrow s_1$ is **DC**, so $s_1 \equiv h\ u_1 \dots u_{m-1}$ for some $u_1, \dots, u_{m-1} \in Pat_\perp$. But then, with a similar reasoning, as $h\ t_1 \dots t_m \not\equiv \perp$ then $s_1\ e_m \equiv h\ u_1 \dots u_{m-1}\ e_m \twoheadrightarrow h\ t_1 \dots t_m$ must be proved applying **DC** at the root, with a proof like the following:

$$\frac{u_1 \twoheadrightarrow t_1 \quad \dots \quad u_{m-1} \twoheadrightarrow t_{m-1} \quad e_m \twoheadrightarrow t_m}{\underbrace{h\ u_1 \dots u_{m-1}}_{s_1}\ e_m \twoheadrightarrow \underbrace{h\ t_1 \dots t_m}_{t}}\ DC$$

Besides, as each $u_i \in Pat_\perp$ and $u_i \twoheadrightarrow t_i$ then $t_i \sqsubseteq u_i$ for $i \in \{1, \dots, m-1\}$. Now, as $size(h\ e_1 \dots e_{m-1} \twoheadrightarrow s_1) < K$ and $s_1 \not\equiv \perp$ then by IH $h\ e_1 \dots e_{m-1} \rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in\ s'_1$ under the conditions stipulated, with $s_1 \equiv h\ u_1 \dots u_{m-1} \sqsubseteq s'_1[\overline{X_1/\perp}]$, so $s'_1 \equiv h\ u'_1 \dots u'_{m-1}$ with $u_i \sqsubseteq u'_i[\overline{X_1/\perp}]$ for $i \in \{1, \dots, m-1\}$. So we can do:

$$\begin{array}{ll} & h\ e_1 \dots e_m \\ \rightarrow^l & (let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \dots u'_{m-1})\ e_m \quad \text{by IH} \\ \rightarrow^l & let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \dots u'_{m-1}\ e_m \quad \text{by (LetAp*)} \end{array}$$

Now we have to do a case distinction over t_m:

a) If $t_m \not\equiv \perp$ then as $size(e_m \twoheadrightarrow t_m) < size(s_1\ e_m \twoheadrightarrow h\ t_1 \dots t_m) \leq K$ we can apply the IH getting $e_m \rightarrow^{l^*} let\ \overline{X_m = a_m}\ in\ t'_m$ under the conditions stipulated. We have two possibilities:

   - If $\overline{X_m} = \emptyset$ then

     $let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \dots u'_{m-1}\ e_m \rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \dots u'_{m-1}\ t'_m$

     and we are done, as $h\ u'_1 \dots u'_{m-1}\ t'_m \in Pat$, $\overline{a_1} \subseteq Exp$ and $|\overline{a_1}| = \overline{\perp}$ by IH, and $t \equiv h\ t_1 \dots t_m \sqsubseteq (h\ u'_1 \dots u'_{m-1}\ t'_m)[\overline{X_1/\perp}]$

because $t_i \sqsubseteq u_i \sqsubseteq u'_i[\overline{X_1}/\ \bot]$ by IH, $t_m \sqsubseteq t'_m$ by IH, and $t'_m \equiv t'_m[\overline{X_1}/\ \bot]$ as $\overline{X_1}$ are were created fresh in a derivation in a position parallel with respect to the position of e_m.

– If $\overline{X_m} \neq \emptyset$ then we can do:

$$\begin{array}{ll} & let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \ldots u'_{m-1}\ e_m \\ \rightarrow^{l^*} & let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \ldots u'_{m-1}(let\ \overline{X_m = a_m}\ in\ t'_m) \quad \text{by IH} \\ \rightarrow^{l} & let\ \overline{X_1 = a_1}\ in \\ & \quad let\ Y = (let\ \overline{X_m = a_m}\ in\ t'_m)\ in\ h\ u'_1 \ldots u'_{m-1}\ Y \quad \text{by (LetIn)} \\ \rightarrow^{l^*} & let\ \overline{X_1 = a_1}\ in \\ & \quad let\ \overline{X_m = a_m}\ in\ let\ Y = t'_m\ in\ h\ u'_1 \ldots u'_{m-1}\ Y \quad \text{by (Flat*)} \\ \rightarrow^{l} & let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_m = a_m}\ in\ h\ u'_1 \ldots u'_{m-1}\ t'_m \quad \text{by (Bind)} \end{array}$$

And we are done, it is very easy to see that all the conditions are fulfilled reasoning in a similar way as we did in the previous case.

b) If $t_m \equiv \bot$ we do a case distinction over e_m:

i) $e_m \in Pat$. Then we are done as $h\ u'_1 \ldots u'_{m-1} \in Pat$ by IH, which implies $h\ u'_1 \ldots u'_{m-1}\ e_m \in Pat$; $\overline{a_1} \subseteq Exp$ and $|\overline{a_1}| = \overline{\bot}$ by IH, and $t \equiv h\ t_1 \ldots t_m \sqsubseteq (h\ u'_1 \ldots u'_{m-1}\ e_m)[\overline{X_1}/\ \bot]$ because $t_i \sqsubseteq u_i \sqsubseteq u'_i[\overline{X_1}/\ \bot]$ by IH, $t_m \equiv \bot \sqsubseteq e_m$, and $e_m \equiv e_m[\overline{X_1}/\ \bot]$ as $\overline{X_1}$ are were created fresh in a derivation in a position parallel with respect to the position of e_m.

ii) $e_m \equiv X_m\ \overline{e_m}$. Then we can do:

$$\begin{array}{ll} & let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \ldots u'_{m-1}\ (X_m\ \overline{e_m}) \\ \rightarrow^{l^*} & let\ \overline{X_1 = a_1}\ in\ let\ Z = X_m\ \overline{e_m}\ in\ h\ u'_1 \ldots u'_{m-1}\ Z \quad \text{by (LetIn)} \end{array}$$

And we are done, it is very easy to see that all the conditions are fulfilled reasoning in a similar way as we did in the previous case, taking account of $|X_m\ \overline{e_m}| = \bot$ and $\bot \sqsubseteq X_m\ \overline{e_m}$.

iii) $e_m \equiv h_m\ \overline{e_m} \notin Pat, h_m \in \Sigma$. Then by lemma 3 we have $h_m\ \overline{e_m} \rightarrow^{l^*} let\ \overline{X_m = a_m}\ in\ h_m\ \overline{t'_m}$ such that $\overline{t'_m} \subseteq Pat$, $\overline{a_m} \subseteq Exp$ and $|\overline{a_m}| = \overline{\bot}$. Then we have two possibilities:

– If $\overline{X_m} = \emptyset$ then it must happen $\overline{e_m} \subseteq Pat$ because otherwise some *let* should be introduced to fulfill lemma 3. So by lemma 15 we have $|h_m\ \overline{e_m}| = \bot$ being $h_m\ \overline{e_m}$ active or junk, and we can do:

$$\begin{array}{ll} & let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \ldots u'_{m-1}\ (h_m\ \overline{e_m}) \\ \rightarrow^{l} & let\ \overline{X_1 = a_1}\ in\ let\ Z = h_m\ \overline{e_m}\ in\ h\ u'_1 \ldots u'_{m-1}\ Z \quad \text{by (LetIn)} \end{array}$$

And we are done, it is very easy to see that all the conditions are fulfilled reasoning in a similar way as we did in the previous case, taking account of $|h_m\ \overline{e_m}| = \bot$ and $\bot \sqsubseteq h_m\ \overline{e_m}$.

– If $\overline{X_m} \neq \emptyset$ then we can do:

$$
\begin{array}{l}
let\ \overline{X_1 = a_1}\ in\ h\ u'_1 \ldots u'_{m-1}\ (h_m\ \overline{e_m}) \\
\rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in \\
\quad h\ u'_1 \ldots u'_{m-1}\ (let\ \overline{X_m = a_m}\ in\ h_m\ \overline{t'_m}) \quad \text{by lemma 3} \\
\rightarrow^{l} let\ \overline{X_1 = a_1}\ in \\
\quad let\ Z = (let\ \overline{X_m = a_m}\ in\ h_m\ \overline{t'_m})\ in \\
\qquad h\ u'_1 \ldots u'_{m-1}\ Z \qquad \text{by (LetIn)} \\
\rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in \\
\quad let\ \overline{X_m = a_m}\ in \\
\qquad let\ Z = h_m\ \overline{t'_m}\ in\ h\ u'_1 \ldots u'_{m-1}\ Z \quad \text{by (Flat*)}
\end{array}
$$

If $h_m\ \overline{t'_m} \notin Pat$ we are done, because as $h_m \in \Sigma$ and $\overline{t'_m} \subseteq Pat$ then $|h_m\ \overline{t'_m}| = \bot$ by lemma 15, and it is very easy to see that all the conditions are fulfilled reasoning in a similar way as we did in the previous case, taking account of $\overline{a_m} \subseteq Exp$ and $|\overline{a_m}| = \overline{\bot}$ by lemma 3, and $\bot \sqsubseteq Z$.

On the other hand, if $h_m\ \overline{t'_m} \in Pat$ we can now do

$$
\begin{array}{l}
let\ \overline{X_1 = a_1}\ in \\
\quad let\ \overline{X_m = a_m}\ in\ let\ Z = h_m\ \overline{t'_m}\ in\ h\ u'_1 \ldots u'_{m-1}\ Z \\
\rightarrow^{l} let\ \overline{X_1 = a_1}\ in \\
\quad let\ \overline{X_m = a_m}\ in\ h\ u'_1 \ldots u'_{m-1}\ (h_m\ \overline{t'_m}) \quad \text{by (Bind)}
\end{array}
$$

And we are done, it is very easy to see that all the conditions are fulfilled reasoning in a similar way as we did in the previous case, taking account of $\bot \sqsubseteq h_m\ \overline{t'_m}$.

OR Then we have case $e \equiv f\ e_1 \ldots e_n\ g_1 \ldots g_m$. In case $n = 0$ we have the following proof:

$$\frac{(r\sigma)g_1 \ldots g_m \twoheadrightarrow t}{f\ g_1 \ldots g_m \twoheadrightarrow t}\ OR$$

for $(f \rightarrow r)\sigma \in [\mathcal{P}]_\bot$. But then we can define σ' as the substitution built from σ replacing every $\bot$ that appears in some expresion in $ran(\sigma)$ with some fresh variable. Then $\sigma \sqsubseteq \sigma'$ and $\sigma' \in PSubst$, so by monotonicity of HOCRWL-derivability, we have $(r\sigma')g_1 \ldots g_m \twoheadrightarrow t$ with a proof of the same size of $(r\sigma)g_1 \ldots g_m \twoheadrightarrow t$, so we can apply the IH getting $(r\sigma')g_1 \ldots g_m \rightarrow^{l^*} let\ \overline{X = a}\ in\ t'$ under the conditions stipulated. But then:

$$
\begin{array}{l}
f\ g_1 \ldots g_m \\
\rightarrow^{l} (r\sigma')\ g_1 \ldots g_m \quad \text{by (Fapp)} \\
\rightarrow^{l} let\ \overline{X = a}\ in\ t' \quad \text{by IH}
\end{array}
$$

and we are done.

In case $n > 0$ we can apply lema 1 to $f\ e_1 \dots e_n\ g_1 \dots g_m \twoheadrightarrow t$ getting:
- $f\ e_1 \dots e_{n-1} \twoheadrightarrow s_1$ for some $s_1 \in Pat_\bot$ and with $size(f\ e_1 \dots e_{n-1} \twoheadrightarrow s_1) < K$.
- $s_1\ e_n\ g_1 \dots g_m \twoheadrightarrow t$ with $size(s_1\ e_n\ g_1 \dots g_m \twoheadrightarrow t) \leq K$.

As $s_1\ e_n\ g_1 \dots g_m \twoheadrightarrow t \not\equiv \bot$ then $s_1 \not\equiv \bot$. But then, as $f \in FS^n$ (because we require that the program rules respect the arity of the functions) and $s_1 \not\equiv \bot$, the only rule that could have been applied at the root of $f\ e_1 \dots e_{n-1} \twoheadrightarrow s_1$ is **DC**, so $s_1 \equiv f\ u_1 \dots u_{n-1}$ for some $u_1, \dots, u_{n-1} \in Pat_\bot$. But then, with a similar reasoning, as $s_1\ e_n\ g_1 \dots g_m \equiv f\ u_1 \dots u_{n-1}\ e_n\ g_1 \dots g_m$ is a total or over application then $s_1\ e_n\ g_1 \dots g_m \twoheadrightarrow t$ must be proved applying **OR** at the root, with a proof like the following:

$$\frac{u_1 \twoheadrightarrow t_1 \quad \dots \quad u_{n-1} \twoheadrightarrow t_{n-1} \quad e_n \twoheadrightarrow t_n \quad (r\sigma)\ g_1 \dots g_m \twoheadrightarrow t}{\underbrace{f\ u_1 \dots u_{n-1}}_{s_1}\ e_n\ g_1 \dots g_m \twoheadrightarrow t}\ OR$$

for $(f\ p_1 \dots p_n \to r)\sigma \in [\mathcal{P}]_\bot$ with $\overline{p}\sigma \equiv \overline{t}$ and $\overline{p} \subseteq Pat$ lineal. Besides, as each $u_i \in Pat_\bot$ and $u_i \twoheadrightarrow t_i$ then $t_i \sqsubseteq u_i$ for $i \in \{1, \dots, n-1\}$. Now, as $size(f\ e_1 \dots e_{n-1} \twoheadrightarrow s_1) < K$ and $s_1 \not\equiv \bot$ then by IH $f\ e_1 \dots e_{n-1} \to^{l^*} let\ \overline{X_1 = a_1}\ in\ s_1'$ under the conditions stipulated, with $s_1 \equiv f\ u_1 \dots u_{n-1} \sqsubseteq s_1'[X_1/\ \bot]$, so $s_1' \equiv f\ u_1' \dots u_{n-1}'$ with $u_i \sqsubseteq u_i'[X_1/\ \bot]$ for $i \in \{1, \dots, m-1\}$. So we can do:

$$\begin{array}{ll} & f\ e_1 \dots e_n\ g_1 \dots g_m \\ \to^l & (let\ \overline{X_1 = a_1}\ in\ f\ u_1' \dots u_{n-1}')\ e_n\ g_1 \dots g_m \quad \text{by IH} \\ \to^l & let\ \overline{X_1 = a_1}\ in\ f\ u_1' \dots u_{n-1}'\ e_n\ g_1 \dots g_m \quad \text{by (LetAp*)} \end{array}$$

Now we have to do a case distinction over t_n:

a) If $t_n \not\equiv \bot$ then as $size(e_n \twoheadrightarrow t_n) < size(s_1\ e_n\ g_1 \dots g_m \twoheadrightarrow t) \leq K$ we can apply the IH getting $e_n \to^{l^*} let\ \overline{X_n = a_n}\ in\ t_n'$ with $t_n \sqsubseteq t_n'[X_n/\ \bot]$, under the conditions stipulated. But then

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ f\ u_1' \dots u_{n-1}'\ e_n\ g_1 \dots g_m \\ \to^{l^*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_n = a_n}\ in\ f\ u_1' \dots u_{n-1}'\ t_n'\ g_1 \dots g_m \end{array}$$

in a similar way as we did in the case for **DC**, using (LetIn), (Flat*) and (Bind) in case $\overline{X_n} \neq \emptyset$. Besides, as $\overline{t} \sqsubseteq u_1, \dots, u_{n-1}, t_n'[\overline{X_n/\ \bot}] \sqsubseteq u_1'[X_1/\ \bot], \dots, u_{n-1}'[X_1/\ \bot], t_n'[X_n/\ \bot] \sqsubseteq u_1', \dots, u_{n-1}', t_n'$ there exists some $\sigma' \in PSubst$ such that $\overline{p}\sigma' \equiv u_1', \dots, u_{n-1}', t_n'$ and $\sigma \sqsubseteq \sigma'$. But then, as $(r\sigma)\ g_1 \dots g_m \twoheadrightarrow t$ then $(r\sigma')\ g_1 \dots g_m \twoheadrightarrow t$ with a proof of the same size. As $size((r\sigma)\ g_1 \dots g_m \twoheadrightarrow t) < size(s_1\ e_n\ g_1 \dots g_m \twoheadrightarrow t) \leq K$ we can apply the IH to $(r\sigma')\ g_1 \dots g_m \twoheadrightarrow t$ getting $(r\sigma')\ g_1 \dots g_m \to^{l^*} let\ \overline{X = a}\ in\ t'$

under the conditions stipulated, so we can chain:

$$\begin{array}{ll} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_n = a_n}\ in\ f\ u'_1 \dots u'_{n-1}\ t'_n\ g_1 \dots g_m & \\ \rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_n = a_n}\ in\ (r\sigma')\ g_1 \dots g_m \dots g_m & \text{by (Fapp)} \\ \rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_n = a_n}\ in\ let\ \overline{X = a}\ in\ t' & \text{by IH} \end{array}$$

And we are done, it is very easy to see that all the conditions are fulfilled.

b) If $t_n \equiv \bot$ we do a case distinction over e_n:

i) $e_n \in Pat$. Then $t_1, \dots, t_{n-1}, \bot \sqsubseteq u'_1, \dots, u'_{n-1}, e_n$ and $u'_1, \dots, u'_{n-1}, e_n \subseteq Pat$ and then there exists some $\sigma' \in PSubst$ to apply (Fapp)with which proceed as we did in the previous case.

ii) $e_n \equiv X_n\ \overline{e_n}$. Then we can do:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ f\ u'_1 \dots u'_{n-1}\ (X_n\ \overline{e_n})\ g_1 \dots g_m \\ \rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in\ let\ Z = X_n\ \overline{e_n}\ in\ f\ u'_1 \dots u'_{n-1}\ Z\ g_1 \dots g_m \end{array}$$

Using (LetIn), (LetAp*). But then $t_1, \dots, t_{n-1}, \bot \sqsubseteq u'_1, \dots, u'_{n-1}, Z$ and $u'_1, \dots, u'_{n-1}, Z \subseteq Pat$, so we can proceed like in the previous case.

iii) $e_n \equiv h_n\ \overline{e_n} \notin Pat, h_n \in \Sigma$. Then we can proceed in a similar way as we did in **DC**, applying lemma 3 to get

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ f\ u'_1 \dots u'_{n-1}\ (h_n\ \overline{e_n})\ g_1 \dots g_m \\ \rightarrow^{l^*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_n = a_n}\ in\ f\ u'_1 \dots u'_{n-1}\ t'_n\ g_1 \dots g_m \end{array}$$

For some $t'_n \in Pat$, so we can proceed like in cases *i)* and *ii)* applying (Fapp) and the HI over $(r\sigma')\ g_1 \dots g_m \twoheadrightarrow t$.

Proof (For Theorem 4, Completeness of HOlet*-rewriting).* Part *(i)* is simply Lemma 4, taking $e' \equiv let\ \overline{X = a}\ in\ t'$. For part *(ii)*, Lemma 4 ensures $t \sqsubseteq t'[\overline{X/\ \bot}]$. But since t is total, it must be the case that $t \equiv t'[\overline{X/\ \bot}]$, and therefore the variables $\overline{X}$ cannot appear in t', and we conclude that $t \equiv t'$. But then we can apply *(Elim)* to $e' \equiv let\ \overline{X = a}\ in\ t$, obtaining $e' \rightarrow^{l} t$, and therefore $e \rightarrow^{l^*} t$.

Proof (of Theorem 5, Equivalence of HOlet*-rewriting and* HOCRWL*-derivability).* This result simply joins together part *(ii)* of Theorems 3 and 4.

4 Higher order *let*-narrowing

Proof (For Lemma 5, Closedness of $\rightarrow^{L}$ under PSubst). The proof is almost identical to proof for the first order version of $\rightarrow^{l}$, just adding a straightforward case for (LetAp), and using the fact that for (LetIn') it does not matter if e_1 was a variable application and $e_1\theta \in Pat$, because we can extract it to a *let* anyway.

Proof (For Theorem 6, Soundness of $\rightsquigarrow^l$ wrt $\rightarrow^L$). It is easier to prove the result for a variant of $\rightsquigarrow^l$ that uses (VNarr) (see Sect. 4) instead of (VAct) and (VBind). Since that variant defines a larger relation, proving soundness for it implies proving soundness also for the original (smaller) relation. The detailed definition for (VNarr) is:

(VNarr) $X\ e \rightsquigarrow^L_{[X/t]}\ t\ (e[X/t])$, for any $t \in Pat$

besides, when combined with (Contx) it must happen that $var(t)$ are fresh and $X \notin BV(\mathcal{C})$.

First we will prove the soundness of narrowing for one step, and see that for one step the lemma is true using $\rightarrow^l$ instead of $\rightarrow^L$, that is, that $e \rightsquigarrow^L_\theta e'$ implies $e\theta \rightarrow^l e'$, if the rule (VNarr) was not used, or $e\theta \rightarrow^{l^0} e'$ if the rule (VNarr) was used. We proceed by a case distinction over the rule used in $e \rightsquigarrow^L_\theta e'$.

(X) Very similar as the first order case, new cases for the rules (LetAp) and (LetIn) are straightforward.

(Narr) Then we have $f\ \bar{t} \rightsquigarrow^L_\theta r\theta$ for $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ fresh, $\theta \in PSubst$ such that $\bar{t}\theta \equiv \overline{p}\theta$. But then $(f\ \overline{p} \rightarrow r)\theta \in [\mathcal{P}]$ and besides $(f\ \overline{p} \rightarrow r)\theta \equiv f\ \overline{p}\theta \rightarrow r\theta \equiv f\ \bar{t}\theta \rightarrow r\theta$, so $(f\ \bar{t} \rightarrow r)\theta \in [\mathcal{P}]$ and we can do $e\theta \equiv f\ \bar{t}\theta \rightarrow^l r\theta \equiv e'$, by (Fapp).

(VNarr) Then we have $X\ a \rightsquigarrow^L_{[X/t]} t\ (a[X/t])$. But then $e\theta \equiv (X\ a)[X/t] \equiv t\ (a[X/t]) \equiv e'$, so $e\theta \rightarrow^{l^0} e\theta \equiv e'$.

(Contxt) Then we have $\mathcal{C}[e] \rightsquigarrow^L_\theta \mathcal{C}\theta[e']$ because $e \rightsquigarrow^L_\theta e'$. Let us do a case distintion over the rule applied in $e \rightsquigarrow^L_\theta e'$:

a) $e \rightsquigarrow^L_\theta e' \equiv f\ \bar{t} \rightsquigarrow^L_\theta r\theta$ by (Narr), for $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ fresh, so $f\ \bar{t}\theta \rightarrow^l r\theta$ by (Fapp), as we saw in the case for (Narr). Then $(\mathcal{C}[e])\theta \equiv (\mathcal{C}[e])\theta|_{\setminus var(\overline{p})}$, because the variables in $var(\overline{p})$ are fresh as $(f\ \overline{p} \rightarrow r)$ is. But then, as $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$ and $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$ by the conditions in (Contx), and $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$ implies $dom(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$, we can apply lemma 11 getting $(\mathcal{C}[e])\theta|_{\setminus var(\overline{p})} \equiv \mathcal{C}\theta|_{\setminus var(\overline{p})}[e\theta|_{\setminus var(\overline{p})}] \equiv \mathcal{C}\theta|_{\setminus var(\overline{p})}[f\ \bar{t}\theta|_{\setminus var(\overline{p})}] \equiv \mathcal{C}\theta[f\ \bar{t}\theta]$, because the variables in $var(\overline{p})$ are fresh again. Besides $vRan(\theta|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$, so we can appply (Contx) combined with an inner (Fapp) step to do $(\mathcal{C}[e])\theta \equiv \mathcal{C}\theta[f\ \bar{t}\theta] \rightarrow^l \mathcal{C}\theta[r\theta] \equiv \mathcal{C}\theta[e']$.

b) $e \rightsquigarrow^L_\theta e' \equiv X\ a \rightsquigarrow^L_{[X/t]} t\ (a[X/t])$ by (VNarr). Then, as $X \notin BV(\mathcal{C})$ and $var(t)$ fresh by the conditions in (Contx), then $dom([X/t]) \cap BV(\mathcal{C}) = vRan([X/t]) \cap BV(\mathcal{C}) = \emptyset$, and so we can apply lemma 11 getting $(\mathcal{C}[e])\theta \equiv (\mathcal{C}[X\ a])[X/t] \equiv \mathcal{C}[X/t][(X\ a)[X/t]] \equiv \mathcal{C}[X/t][t\ (a[X/t])] \equiv \mathcal{C}\theta[e']$.

c) In case a different rule was applied in $e \rightsquigarrow^L_\theta e'$ then $\theta = \epsilon$. Besides, by the proof of the other cases we have $e \equiv e\epsilon \equiv e\theta \rightarrow^l e'$, so $(\mathcal{C}[e])\theta \equiv (\mathcal{C}[e])\epsilon \equiv \mathcal{C}[e] \rightarrow^l \mathcal{C}[e'] \equiv \mathcal{C}\epsilon[e'] \equiv \mathcal{C}\theta[e']$.

Now we will try to prove the lemma for any number of steps $\rightarrow^l$, proceeding by induction over the length n of $e \leadsto_\theta^{L^n} e'$. The case $e \leadsto_\epsilon^{L^0} e \equiv e'$ is straightforward because $e \rightarrow^{l^0} e \equiv e'$. The problem comes for $e \leadsto^L_\sigma e'' \leadsto_\gamma^{L^n} e'$, when trying to chain the induction hypothesis: by IH $e''\gamma \rightarrow^{l^*} e'$, and by the proof for one step $e\sigma \rightarrow^{l^=} e''$, but as $\rightarrow^l$ is not closed under $PSubst$ we cannot chain these steps. But as $\rightarrow^l \subseteq \rightarrow^L$, then we have covered the base case for $\rightarrow^L$, and for the inductive step we have $e''\gamma \rightarrow^{L^*} e'$ and $e\sigma \rightarrow^{l^=} e''$. But then $e\sigma \rightarrow^{l^=} e''$ implies $e\sigma\gamma \rightarrow^{l^=} e''\gamma$, by lemma 5, so we can chain $e\sigma\gamma \rightarrow^{l^=} e''\gamma \rightarrow^{L^*} e'$.

Proof (For theorem 7).

a) If $e \leadsto_\theta^{L^*} e'$ then $e\theta \rightarrow^{L^*} e'$, by theorem 6, but then $[\![e']\!] \subseteq [\![e\theta]\!]$ by theorem 3.
b) If $e \leadsto_\theta^{L^*} t$ then $[\![t]\!] \subseteq [\![e\theta]\!]$ by *a)*. But then as $t \in Pat$ implies $t \in [\![t]\!]$ then $t \in [\![e\theta]\!]$, so $e\theta \rightarrow^{l^*} t$ by theorem 4

Proof (For lemma 6). Let us proof the lemma for one narrowing step first, we will see that if $e\theta \rightarrow^l e'$ then $e \leadsto^l_\sigma e'$ under the conditions above. Let us do a case distinction over the rule applied in $e\theta \rightarrow^l e'$:

X $\in \{Elim, Flat, LetIn, LetAp\}$ In this case lifting can be done very easily using $\sigma = \epsilon$ and $\theta' = \theta$, just being a little careful in the case for (LetIn):
- If $e_2\theta$ is a variable application or a *let*-rooted expression then e_2 must be also a variable application or *let*-rooted expression.
- If $e_2\theta$ is active or junk then it might happen:
  a) $e_2 \equiv h\ \overline{t_2}$: Then $h\ \overline{t_2}$ is also junk or active because h in $e_2\theta$ has the same arity and is applied to the same number of arguments.
  b) $e_2 \equiv Y\ \overline{t_2}$: The only problematic case is that in which $\overline{t_2} = \emptyset$ and so $e_2 \equiv Y \in Pat$. But the only way it could happen with $e_2\theta$ being active or junk, is that $\theta = [Y/e']$ for some e' active or junk, but then $\theta \notin PSubst$, which contradicts the hypothesis of the lifting lemma.

Bind Then $e\theta \equiv let\ X = s_1\theta\ in\ s_2\theta \rightarrow^l s_2\theta[X/s_1\theta] \equiv e'$ with $s_1\theta \in Pat$. If $s_1 \in Pat$ also then the proof is straightforward and similar to the previous case. Otherwise we will need the following lemma:

Lemma 16. *Given $e \in LExp \setminus Pat, \theta \in PSubst$ if one has $e\theta \in Pat$ then $e \in Exp$ and $\forall o \in \mathcal{O}(e)$ such that $e|_o \equiv X\ a_1 \dots a_k$ with $k > 0$ then $\theta(X) \equiv h\ t_1 \dots t_m$ with $h \in CS^n, m + k \leq n$ or $h \in FS^n, m + k < n$.*

Proof (Sketch). $e \in Exp$ because *let* expression do not disappear after applying substitutions. Using proof by contradiction, if $\theta(X) \equiv h\ t_1 \dots t_m$ does not hold under those conditions for some X then some subexpression of $e\theta$ is not a pattern.

Let $s_1 \equiv s_1[Z_1\ \overline{a_1}, \dots, Z_n\ \overline{a_n}]$ be[6]. As $s_1\theta \in Pat$, then by lemma 16 we have $\forall Z_i \in \{Z_1, \dots, Z_n\}, \theta(Z_i) \equiv h_i\ \overline{t_i}$. Now we define $\sigma(Z_i) \equiv h_i\ \overline{U_i}$ for $\overline{U_i}$ fresh and linear, so:

[6] With $s_1[\ , \dots,\ ]$ we denote a context with several holes, defined as usual.

- $dom(\sigma) = \overline{Z} \subseteq FV(s_1) = FV(s_1) \setminus \{X\}$, because X cannot appear in s_1 because there are no recursive *lets*.
- $Ran(\sigma)$ contains only fresh shallow patterns. $vRan(\sigma) = \overline{U}$, where $\overline{U}$ is just an alias for $\bigcup_i \overline{U_i}$.
- $s_1\sigma \equiv s_1[h_1\ \overline{U_1}\ \overline{a_1}, \ldots, h_n\ \overline{U_n}\ \overline{a_n}] \in Pat$
- $Z_i\sigma[\overline{U_i}/t_i] \equiv h_i\ \overline{U_i}[\overline{U_i}/t_i] \equiv h_i\overline{t_i} \equiv Z_i\theta$.

So we can apply (VBind) to do:

$$e \equiv let\ X = s_1\ in\ s_2 \rightsquigarrow^{l}_{\sigma} s_2\sigma[X/s_1\sigma] \equiv e''$$

Let $\theta'_1 = \biguplus_i [\overline{U_i/t_i}] \in PSubst$. Then $dom(\theta'_1) = \overline{U}$ and $\sigma\theta'_1 = \theta[\overline{Z}]$, as we saw before. Now we can define $\theta'_0 = \theta$ and $\theta' = \theta'_0 \uplus \theta'_1$, which is correctly defined as $dom(\theta'_0) = dom(\theta) \subseteq \mathcal{W}$, and $dom(\theta'_1) = \overline{U}$ which are fresh. Now we will see how the conditions in lemma 6 hold:

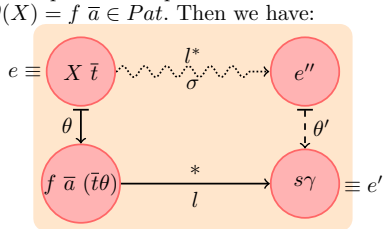
- Condition *ii)* : $\sigma\theta' = \theta[\mathcal{W}]$: Given $Y \in \mathcal{W}$
  a) If $Y \in \overline{Z}$: Then $Y\theta \equiv Y\sigma\theta'_1$ because $\sigma\theta'_1 = \theta[\overline{Z}]$. As $Y \in \overline{Z}$ then $Y \in dom(\sigma)$ and $var(Y\sigma)$ are fresh and do not appear in $dom(\theta'_0) = dom(\theta)$. But then $Y\sigma\theta'_1 = Y\sigma\theta$.
  b) If $Y \in \mathcal{W} \setminus \overline{Z}$: Then $Y \notin dom(\sigma)$ and $var(Y\sigma) \cap dom(\theta'_1) = \emptyset$ by definition, so by definition of θ'_0 we have $Y\theta \equiv Y\theta'_0 \equiv Y\theta' \equiv Y\sigma\theta'$.
- Condition *i)* : $e''\theta' \equiv e'$: By the variable convention $X \notin (dom(\theta) \cup \overline{vRan(\theta)})$, so we can apply lemma 10 over $e' \equiv s_2\theta[X/s_1\theta] \equiv s_2[X/s_1]\theta$. We have seen $X \notin dom(\sigma)$ and besides $X \notin vRan(\sigma)$, so we also can apply lemma 10 to get $e'' \equiv s_2\sigma[X/s_1\sigma] \equiv s_2[X/s_1]\sigma$. But $\mathcal{W} \supseteq FV(e) = FV(let\ X = s_1\ in\ s_2) = FV(s_1) \cup (FV(s_2) \setminus \{X\}) \supseteq FV(s_2[X/s_1])$. So, as $\sigma\theta' = \theta[\mathcal{W}]$, we have:

$$\underbrace{s_2[X/s_1]\sigma}_{e''}\,\theta' \equiv \underbrace{s_2[X/s_1]\theta}_{e'}$$

- Condition *iii)* : $(dom(\theta') \cup vRan(\theta')) \cap \mathcal{B} = \emptyset$: Remember $\theta' = \theta'_0 \uplus \theta'_1$. Regarding θ'_0 the condition holds as $\theta'_0 = \theta$, and $(dom(\theta) \cup vRan(\theta)) \cap \mathcal{B} = \emptyset$ by the hypothesis. Regading θ'_1, if $Y \in dom(\theta'_1) = \overline{U}$ then Y is fresh and so $Y \notin \mathcal{B}$. On the other hand, $vRan(\theta'_1) = var(\bigcup_i \overline{t_i}) \subseteq vRan(\theta)$, and $vRan(\theta) \cap \mathcal{B} = \emptyset$ by the hypothesis, so $vRan(\theta'_1) \cap \mathcal{B} = \emptyset$.

Fapp Then e can have two possible shapes:

1. $e \equiv X\ \overline{t}$, with $\theta(X) = f\ \overline{a} \in Pat$. Then we have:



With an (Fapp) step $e\theta \equiv f\ \overline{a}\ (\overline{t}\theta) \rightarrow^{l} s\gamma$ using a fresh variant $(f\ \overline{p} \rightarrow s) \in \mathcal{P}$ such that $(X\ \overline{t})\theta \equiv (f\ \overline{p})\gamma$ for some $\gamma \in PSubst$. We can assume that $dom(\gamma) \subseteq FV(f\ \overline{p} \rightarrow s)$ without loss of generality. But then $dom(\theta) \cap dom(\gamma) = \emptyset$, and so $\theta \uplus \gamma$ is correctly defined, and it is a unifier of $X\ \overline{t}$ and $f\ \overline{p}$. So, there must exists $\sigma = mgu(X\ \overline{t}, f\ \overline{p})$, which we can use to perform a (VAct) step, because $\theta \in PSubst$ and $(X\ \overline{t})\theta$ active imply $\overline{t} \neq \emptyset$:

$$e \equiv X\ \overline{t} \rightsquigarrow^{l}_{\sigma}\ s\sigma \equiv e''$$

As this unifier is an mgu then $dom(\sigma) \subseteq FV(X\ \overline{t}) \cup FV(f\ \overline{p})$, $vRan(\sigma) \subseteq FV(X\ \overline{t}) \cup FV(f\ \overline{p})$ and $\sigma \lesssim (\theta \uplus \gamma)$, so there must exists $\theta'_1 \in PSubst$ such that $\sigma\theta'_1 = \theta \uplus \gamma$. Besides we can define $\theta'_0 = \theta|_{\setminus (dom(\theta'_1) \cup FV(X\ \overline{t}))}$ and then we can take $\theta' = \theta'_0 \uplus \theta'_1$ which is correctly defined as obviously $dom(\theta'_0) \cap dom(\theta'_1) = \emptyset$. Besides $dom(\theta'_0) \cap (FV(X\ \overline{t}) \cup FV(f\ \overline{p})) = \emptyset$, as if $Y \in FV(X\ \overline{t})$ then $Y \notin dom(\theta'_0)$ by definition; and if $Y \in FV(f\ \overline{p})$ then $Y \notin dom(\theta)$ as $\overline{p}$ belong to the fresh variant, and so $Y \notin dom(\theta'_0)$. Then the conditions in lemma 6 hold:

- Condition *i)* : $e''\theta' \equiv e'$: As $e''\theta' \equiv s\sigma\theta' \equiv s\sigma\theta'_1$ because given $Y \in \overline{FV(s\sigma)}$, if $Y \in FV(s)$ then it belongs to the fresh variant and so $Y \notin dom(\theta) \supseteq dom(\theta'_0)$; and if $Y \in vRan(\sigma) \subseteq FV(X\ \overline{t}) \cup FV(f\ \overline{p})$ then $Y \notin dom(\theta'_0)$ because $dom(\theta'_0) \cap (FV(X\ \overline{t}) \cup FV(f\ \overline{p})) = \emptyset$. But $s\sigma\theta'_1 \equiv s(\theta \uplus \gamma) \equiv s\gamma \equiv e'$, because $\sigma\theta'_1 = \theta \uplus \gamma$ and s is part of the fresh variant.
- Condition *ii)* : $\sigma\theta' = \theta[\mathcal{W}]$: Given $Y \in \mathcal{W}$, if $Y \in FV(X\ \overline{t})$ then $\overline{Y \notin dom(\gamma)}$ and so $Y\theta \equiv Y(\theta \uplus \gamma) \equiv Y\sigma\theta'_1$, as $\sigma\theta'_1 = \theta \uplus \gamma$. But $Y\sigma\theta'_1 \equiv Y\sigma\theta'$ because given $Z \in var(Y\sigma)$, if $Z \equiv Y$ then as $Y \in FV(X\ \overline{t})$ then $Z \equiv Y \notin dom(\theta'_0)$ by definition of θ'_0; if $Z \in vRan(\sigma)$ then $Z \notin dom(\theta'_0)$, as we saw before.
  On the other hand, $(\mathcal{W} \setminus FV(X\ \overline{t})) \cap (FV(X\ \overline{t}) \cup FV(f\ \overline{p})) = (\mathcal{W} \setminus FV(X\ \overline{t}) \cap FV(X\ \overline{t})) \cup (\mathcal{W} \setminus FV(X\ \overline{t}) \cap FV(f\ \overline{p})) = \emptyset \cup \emptyset = \emptyset$, because $FV(f\ \overline{p})$ are part of the fresh variant. So, if $Y \in \mathcal{W} \setminus FV(X\ \overline{t})$, then $Y \notin dom(\sigma) \subseteq FV(X\ \overline{t}) \cup FV(f\ \overline{p})$. Now if $Y \in dom(\theta'_0)$ then $Y\theta \equiv Y\theta'_0$ (by definition of θ'_0), $Y\theta'_0 \equiv Y\theta'$ (as $Y \in dom(\theta'_0)$), $Y\theta' \equiv Y\sigma\theta'$ (as $Y \notin dom(\sigma)$). If $Y \in dom(\theta'_1)$, $Y\theta \equiv Y(\theta \uplus \gamma)$ (as $Y \in \mathcal{W} \setminus FV(X\ \overline{t})$ implies it does not appear in the fresh instance), $Y(\theta \uplus \gamma) \equiv Y\sigma\theta'_1$ (as $\sigma\theta'_1 = \theta \uplus \gamma$), $Y\sigma\theta'_1 \equiv Y\theta'_1$ (as $Y \notin dom(\sigma)$), $Y\theta'_1 \equiv Y\theta'$ (as $Y \in dom(\theta'_1)$) and $Y\theta' \equiv Y\sigma\theta'$ (as $Y \notin dom(\sigma)$). And if $Y \notin (dom(\theta'_0) \cup dom(\theta'_1))$ then $Y \notin dom(\theta')$, and as $Y \notin dom(\sigma)$ and $Y\theta \equiv Y(\theta \uplus \gamma)$, then $Y\theta \equiv Y(\theta \uplus \gamma) \equiv Y\sigma\theta'_1 \equiv Y \equiv Y\sigma\theta'$.
- Condition *iii.1)* : $dom(\theta') \cap \mathcal{B} = \emptyset$. Remember $\theta' = \theta'_0 \uplus \theta'_1$:
  - $dom(\theta'_0) \cap \mathcal{B} = \emptyset$: Given $Y \in dom(\theta'_0)$ then $Y \in dom(\theta)$ by definition of θ'_0, and so $Y \notin \mathcal{B}$, because $dom(\theta) \cap \mathcal{B} = \emptyset$ by hypothesis.
  - $dom(\theta'_1) \cap \mathcal{B} = \emptyset$: As σ is an mgu and $\sigma \lesssim \theta \uplus \gamma$, then $dom(\sigma) \subseteq dom(\theta \uplus \gamma)$. Given $Z \in \mathcal{B}$ then $Z \notin dom(\theta)$, as $dom(\theta) \cap \mathcal{B} = \emptyset$ by hypothesis, and $Z \notin dom(\gamma) \subseteq FV(f\ \overline{p} \rightarrow s)$ which are fresh, so

$Z \notin dom(\sigma)$. But then, as $\sigma\theta'_1 = \theta \uplus \gamma$, $Z \equiv Z(\theta \uplus \gamma) \equiv Z\sigma\theta'_1 \equiv Z\theta'_1$, so $Z \notin dom(\theta'_1)$.

- Condition *iii.2)* : $vRan(\theta') \cap \mathcal{B} = \emptyset$. Remember $\theta' = \theta'_0 \uplus \theta'_1$:
  - $vRan(\theta'_0) \cap \mathcal{B} = \emptyset$: Given $Y \in dom(\theta'_0)$ then $Y\theta'_0 \equiv Y\theta$ by definition of θ'_0. As $vRan(\theta) \cap \mathcal{B} = \emptyset$ by hypothesis then it must happen $var(Y\theta) \cap \mathcal{B} = \emptyset$, so $var(Y\theta'_0) \cap \mathcal{B} = \emptyset$.
  - $vRan(\theta'_1) \cap \mathcal{B} = \emptyset$: As $\sigma\theta'_1 = \theta \uplus \gamma$ then we can assume $dom(\theta'_1) \subseteq vRan(\sigma) \cup (dom(\theta \uplus \gamma) \setminus dom(\sigma))$.
    - Let $X \in dom(\theta'_1) \cap vRan(\sigma)$ be such that $X\theta'_1 \equiv r[Z]$ with $Z \in \mathcal{B}$. We will see that this $Z \in \mathcal{B}$ cannot appear in $X\theta'_1$ without leading to contradiction. The intuition is, as $vRan(\theta) \cap \mathcal{B} = \emptyset$ and $vRan(\gamma|_{vExtra(R)}) \cap \mathcal{B} = \emptyset$, then every $Z \in \mathcal{B}$ must come from anappearance in e of the same variable, transmitted to e' by the matching substitution γ, and so transmitted to e'' by σ.
      As $X \in vRan(\sigma)$ then there must exists $Y \in dom(\sigma)$ such that $Y \longmapsto^{\sigma} s_1[X]_p \longmapsto^{\theta'_1} s_2[r[Z]]_p$. But as $\sigma\theta'_1 = \theta \uplus \gamma$ then $Y \longmapsto^{\theta \uplus \gamma} s_2[r[Z]]_p$. Then, $Z \in vRan(\theta \uplus \gamma)$, but $Z \in \mathcal{B}, vRan(\theta) \cap \mathcal{B} = \emptyset, vRan(\gamma|_{vExtra(R)}) \cap \mathcal{B} = \emptyset, dom(\gamma) \subseteq FV(f\ \overline{p} \to s)$, so it must happen $Z \in vRan(\gamma|_{FV(\overline{p})})$, and as a consequence $Y \in FV(\overline{p})$. Let $o \in \mathcal{O}(f\ \overline{p})$ (set of positions in $f\ \overline{p}$) be such that $f\ \overline{p}|_o \equiv Y$, then:
      - $((X\ \overline{t})\sigma)|_o \equiv ((f\ \overline{p})\sigma)|_o \equiv ((f\ \overline{p})|_o)\sigma \equiv Y\sigma \equiv s_1[X]_p$.
      - As $X\ \overline{t} \notin dom(\gamma)$, which are the fresh variables of the variant of the program rule, $((X\ \overline{t})\theta)|_o \equiv ((X\ \overline{t})(\theta \uplus \gamma))|_o \equiv ((f\ \overline{p})(\theta \uplus \gamma))|_o \equiv ((f\ \overline{p})|_o)(\theta \uplus \gamma) \equiv Y(\theta \uplus \gamma) \equiv s_2[r[Z]]_p$

      So, as $X \in dom(\theta'_1)$ then $X \notin \mathcal{B}$ and $Z \in \mathcal{B}$ has been introduced by θ, but this is impossible as $vRan(\theta) \cap \mathcal{B} = \emptyset$.
    - Let $Y \in dom(\theta) \setminus dom(\sigma)$ be. Then $Y\theta \equiv Y(\theta \uplus \gamma)$ (as $Y \in dom(\theta)$), $Y(\theta \uplus \gamma) \equiv Y\sigma\theta'_1$ (as $\sigma\theta'_1 = \theta \uplus \gamma$), $Y\sigma\theta'_1 \equiv Y\theta'_1$ (as $Y \notin dom(\sigma)$. But then no var in $\mathcal{B}$ can appear in $Y\theta'_1 \equiv Y\theta$ as $(dom(\theta) \cup vRan(\theta)) \cap \mathcal{B} = \emptyset$.
    - Let $Y \in dom(\gamma) \setminus dom(\sigma)$ be. Then $Y\gamma \equiv Y(\theta \uplus \gamma) \equiv Y\sigma\theta'_1 \equiv Y\theta'_1$, reasoning like in the previous case. As $dom(\gamma) \subseteq FV(f\ \overline{p} \to s)$ it can happen:
      - $Y \notin FV(f\ \overline{p})$: Then no var in $\mathcal{B}$ can appear in $Y\gamma$ because $vRan(\gamma|_{vExtra(R)}) \cap \mathcal{B} = \emptyset$ by the hypothesis.
      - $Y \in FV(f\ \overline{p})$: Let $Z \in \mathcal{B}$ appearing in $Y\gamma$, then Z appears in $f\ \overline{t}$ because $(f\ \overline{p})\gamma \equiv (f\ \overline{t})\theta$ and $vRan(\theta) \cap \mathcal{B} = \emptyset$. So it must happen $Y \in dom(\sigma)$ because otherwise σ could not be a unifier of $(f\ \overline{t})$ and $(f\ \overline{p})$ as Y is part of the fresh instance and so it cannot belong to $\mathcal{B}$. But this is a contradiction so this case is impossible.

2. $e \equiv f\ \overline{t}$. Then we can proceed in a similar way as we did in the previous case, but usin (Narr) instead of (VAct).

Contx Then we have $e \equiv \mathcal{C}[s]$. By hypothesis $(dom(\theta) \cup vRan(\theta)) \cap \mathcal{B} = \emptyset$ and $BV(\mathcal{C}[s]) \subseteq \mathcal{B}$, so by lemma 11 $e\theta \equiv (\mathcal{C}[s])\theta \equiv \mathcal{C}\theta[s\theta]$, and the step was

$$e\theta \equiv \mathcal{C}\theta[s\theta] \rightarrow^{l} \mathcal{C}\theta[s'] \equiv e', \text{ because } s\theta \rightarrow^{l} s'$$

Then we know that the lemma holds for $s\theta \rightarrow^{l} s'$, by the proof of the other cases, so taking $\mathcal{W}' = \mathcal{W} \cup FV(s)$ and $\mathcal{B}' = \mathcal{B}$ (as $BV(s) \subseteq BV(\mathcal{C}[s])$) we can do $s \rightsquigarrow^{l}_{\sigma_2} s''$ for some θ'_2 under the conditions stipulated. Now we can put this step into (Contx) to do:

$$e \equiv \mathcal{C}[s] \rightsquigarrow^{l}_{\sigma_2} \mathcal{C}\sigma_2[s''] \equiv e'' \text{ taking } \sigma = \sigma_2 \text{ and } \theta' = \theta'_2$$

because:

- If $s \rightsquigarrow^{l}_{\sigma_2} s''$ was a (Narr) or (VAct) step which lifts a (Fapp) step that uses the fresh variant $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ and adjusts with $\gamma \in PSubst$, then:
  - $dom(\sigma_2) \cap BV(\mathcal{C}) = \emptyset$: As $\sigma_2 = mgu(s, f\ \overline{p})$ then $dom(\sigma_2) \subseteq FV(s) \cup FV(f\ \overline{p})$. As $\sigma_2 \lesssim \theta \uplus \gamma$ and it is an mgu then $dom(\sigma_2) \subseteq dom(\theta \uplus \gamma)$. If $X \in FV(s) \cap dom(\sigma_2)$ then $X \notin dom(\gamma) \subseteq FV(f\ \overline{p} \rightarrow r)$, so it must happen $X \in dom(\theta)$; but then $X \notin BV(\mathcal{C})$ because $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$ by the variable convention.
    Otherwise it could happen $X \in FV(f\ \overline{p}) \cap dom(\sigma_2)$, then X appears in the fresh variant and so it cannot appear in $\mathcal{C}$.
  - $vRan(\sigma_2|_{\setminus var(\overline{p})}) \cap BV(\mathcal{C}) = \emptyset$: As $dom(\sigma_2) \subseteq FV(s) \cup FV(f\ \overline{p})$ then $vRan(\sigma_2|_{\setminus var(\overline{p})}) = vRan(\sigma_2|_{FV(s)})$. But as $\sigma_2 = mgu(s, f\ \overline{p})$ then $vRan(\sigma|_{FV(s)}) \subseteq FV(f\ \overline{p})$, which are part of the fresh variant, so every variable in $vRan(\sigma_2|_{\setminus var(\overline{p})})$ is fresh and so cannot appear in $\mathcal{C}$.
- If $s \rightsquigarrow^{l}_{\sigma_2} s''$ was a (VBind) step then:
  - $dom(\sigma_2) \cap BV(\mathcal{C}) = \emptyset$: As $dom(\sigma_2) = \overline{Z} \subseteq dom(\theta)$, and $dom(\theta) \cap BV(\mathcal{C}) = \emptyset$ as we saw before.
  - $vRan(\sigma_2) \cap BV(\mathcal{C}) = \emptyset$: Because $vRan(\sigma_2)$ only contains fresh patterns.

Then the conditions in lemma 6 hold:

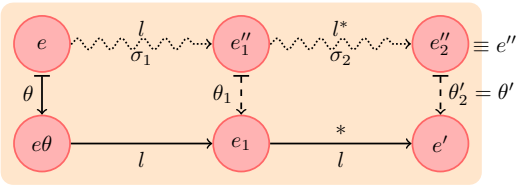
- Condition *ii)* : $\sigma\theta' = \theta[\mathcal{W}]$: Because $\mathcal{W} \subseteq \mathcal{W}'$, and $\sigma_2\theta'_2 = \theta[\mathcal{W}']$, by the proof of the other cases.
- Condition *i)* : $e''\theta' \equiv e'$: As $BV(\mathcal{C}\sigma_2) = BV(\mathcal{C})$, by the variable convention, $BV(\mathcal{C}) \subseteq BV(e) \subseteq BV(\mathcal{B})$, by the hypothesis, and $(dom(\theta'_2) \cup vRan(\theta'_2)) \cap \mathcal{B} = \emptyset$, by the proof of the other cases, then $(dom(\theta'_2) \cup vRan(\theta'_2)) \cap BV(\mathcal{C}\sigma_2) = \emptyset$. But then:

$$e''\theta' \equiv (\mathcal{C}\sigma_2[s''])\theta'_2 \equiv \underbrace{\mathcal{C}\sigma_2\theta'_2}_{\mathcal{C}\theta}[\underbrace{s''\theta'_2}_{s'}] \equiv e'$$

Because we have $s''\theta'_2 \equiv s'$, by the proof of the other cases, and because $FV(\mathcal{C}) \subseteq FV(e) \subseteq \mathcal{W}$ and $\sigma_2\theta'_2 = \theta[\mathcal{W}]$, as we saw in the previous case (remember $\sigma = \sigma_2$ and $\theta' = \theta'_2$).

– Condition *iii)* : $(dom(\theta') \cup vRan(\theta')) \cap \mathcal{B} = \emptyset$: Because $\theta' = \theta'_2$ and the proof of the other cases.

Now we will prove the lemma for any number of steps proceeding by induction over the number n of steps of the derivation $e\theta \rightarrow^{l^n} e'$. The base case where $n = 0$ is straightforward, as then we have $e\theta \rightarrow^{l^0} e\theta \equiv e'$ so we can do $e \rightsquigarrow^{l^0}_{\epsilon} e \equiv e''$, so $\sigma = \epsilon$ and taking $\theta' = \theta$ the lemma holds. In the inductive step we have $e\theta \rightarrow^{l} e_1 \rightarrow^{l^*} e'$, and we will try to build the following diagram:



By the proof for one step we have $e \rightsquigarrow^{l}_{\sigma_1} e''_1$ and $\theta'_1 \in PSubst$ under the conditions stipulated. In order to deal with the IH we define the sets $\mathcal{B}_1 = \mathcal{B} \cup BV(e_1)$ and $\mathcal{W}_1 = (\mathcal{W} \setminus dom(\sigma_1)) \cup vRan(\sigma_1) \cup vE$, where vE is the set of extra variables in the fresh variant $f\ \overline{p} \rightarrow s$ used in $e \rightsquigarrow^{l}_{\sigma_1} e''_1$, if it was a (Narr) or (VAct) step; $vE = \overline{U} = vRan(\sigma_1)$, if it was a (VBind) step; or vE is empty otherwise. We also define $\theta_1 = \theta'_1|_{\mathcal{W}_1}$. Then:

– $FV(e''_1) \cup dom(\theta_1) \subseteq \mathcal{W}_1$: We have $dom(\theta_1) \subseteq \mathcal{W}_1$ by definition of θ_1. On the other hand we can prove $FV(e''_1) \subseteq \mathcal{W}_1$ just reasoning as we did in the first order version of this lemma, just adding a case for $X \notin FV(e)$ and X introduced by a (VBind) step, in which $X \in vE$, and hence $X \in \mathcal{W}_1$.
– $e''_1\theta_1 \equiv e_1$: Because as we have seen, $FV(e''_1) \subseteq \mathcal{W}_1$, and so $e''_1\theta_1 \equiv e''_1\theta'_1|_{\mathcal{W}_1} \equiv e''_1\theta'_1 \equiv e_1$, by the proof for one step.
– $BV(e''_1) \subseteq \mathcal{B}_1$: As $\theta'_1 \in PSubst$, $e''_1\theta'_1 \equiv e_1$ and no $PSubst$ can introduce any binding then $BV(e_1) = BV(e''_1)$. But $\mathcal{B}_1 = \mathcal{B} \cup BV(e_1)$, so $BV(e''_1) = BV(e_1) \subseteq \mathcal{B}_1$.
– $(dom(\theta_1) \cup vRan(\theta_1)) \cap \mathcal{B}_1 = \emptyset$: As $\theta'_1 \in PSubst$, $e''_1\theta'_1 \equiv e_1$ and no $PSubst$ can introduce any binding then $BV(e_1) = BV(e''_1)$. Then it can happen:
  a) $BV(e''_1) \subseteq BV(e)$: Then $\mathcal{B} = \mathcal{B}_1$, as $BV(e_1) = BV(e''_1) \subseteq BV(e) \subseteq \mathcal{B}$ by hypothesis. Then, as $(dom(\theta'_1) \cup vRan(\theta'_1)) \cap \mathcal{B} = \emptyset$ by the proof for one step, then $(dom(\theta'_1) \cup vRan(\theta'_1)) \cap \mathcal{B}_1 = \emptyset$, and so $(dom(\theta_1) \cup vRan(\theta_1)) \cap \mathcal{B}_1 = \emptyset$, because $\theta_1 = \theta'_1|_{\mathcal{W}_1}$ and so its domain and variable range is smaller than the domain of θ'_1.
  b) $BV(e''_1) \supset BV(e)$: Then $e \rightsquigarrow^{l}_{\sigma_1} e''_1$ must have been a (LetIn) step and so $\sigma = \epsilon$ and $\theta'_1 = \theta$. As the new bounded variable Z is fresh wrt θ then it is also fresh for $\theta'_1 = \theta$, and so $\mathcal{B}_1 = \mathcal{B} \cup \{Z\}$ has no intersection with $dom(\theta'_1) \cup vRan(\theta'_1)$ nor with $dom(\theta_1) \cup vRan(\theta_1)$, which is smaller.
– For each instance of program rule $R\mu \in [\mathcal{P}]$ used in an (Fapp) step it happens $vRan(\mu|_{vExtra(R)}) \cap \mathcal{B}_1 = \emptyset$. As we have seen in the previuos case either $\mathcal{B}_1 = \mathcal{B}$ or $\mathcal{B}_1 = \mathcal{B} \cup \{Z\}$ for some Z fresh, so we can assume without loss of generality that for any of those μ we have $Z \notin vRan(\mu|_{vExtra(R)})$.

– $\sigma_1\theta_1 = \theta[\mathcal{W}]$: It is enough to see that $\sigma_1\theta_1 = \sigma_1\theta'_1[\mathcal{W}]$, because we have $\sigma_1\theta'_1 = \theta[\mathcal{W}]$ by the proof for one step, and this is true because given $X \in \mathcal{W}$:
  a) If $X \in dom(\sigma_1)$ then $FV(X\sigma_1) \subseteq vRan(\sigma_1) \subseteq \mathcal{W}_1$, so as $\theta_1 = \theta'_1|_{\mathcal{W}_1}$ then $X\sigma_1\theta_1 \equiv X\sigma_1\theta'_1|_{\mathcal{W}_1} \equiv X\sigma_1\theta'_1$.
  b) If $X \in \mathcal{W}\backslash dom(\sigma_1)$ then $X \in \mathcal{W}_1$ by definition, and so $X\sigma_1\theta_1 \equiv X\theta_1$ (as $X \notin dom(\sigma_1)$), $X\theta_1 \equiv X\theta'_1|_{\mathcal{W}_1} \equiv X\theta'_1$ (as $X \in \mathcal{W}_1$), and $X\theta'_1 \equiv X\sigma\theta'_1$ (as $X \notin dom(\sigma_1)$).

So we have $e''_1\theta_1 \equiv e_1$ and $e_1 \to^{l^*} e'$, but then we can apply the induction hypothesis to $e''_1\theta_1 \to^{l^*} e'$ using $\mathcal{W}_1$ and $\mathcal{B}_1$, which fulfil the hypothesis of the lemma, as we have seen. Then we get $e''_1 \rightsquigarrow^{l^*}_{\sigma_2} e''_2$ and $\theta'_2 \in PSubst$ under the conditions stipulated. But then we have:

$$e \rightsquigarrow^{l}_{\sigma_1} e''_1 \rightsquigarrow^{l^*}_{\sigma_2} e''_2 \text{ taking } e'' \equiv e''_2,\ \sigma = \sigma_1\sigma_2 \text{ and } \theta' = \theta'_2$$

for which we can prove:

– Condition *i)* : $e''\theta' \equiv e'$: As $e''\theta' \equiv e''_2\theta'_2 \equiv e'$ by IH.
– Condition *ii)* : $\sigma\theta' = \theta[\mathcal{W}]$: That is, $\sigma_1\sigma_2\theta'_2 = \theta[\mathcal{W}]$. As we have $\sigma_1\theta_1 = \theta[\mathcal{W}]$, as we saw before, all that is left is proving $\sigma_1\sigma_2\theta'_2 = \sigma_1\theta_1[\mathcal{W}]$, which happens because given $X \in \mathcal{W}$:
  a) If $X \in dom(\sigma_1)$ then $FV(X\sigma_1) \subseteq vRan(\sigma_1) \subseteq \mathcal{W}_1$, so as $\sigma_2\theta'_2 = \theta_1[\mathcal{W}_1]$ by IH, then $(X\sigma_1)\sigma_2\theta'_2 \equiv (X\sigma_1)\theta_1$.
  b) If $X \in \mathcal{W}\backslash dom(\sigma_1)$ then $X \in \mathcal{W}_1$ by definition, and so, as $\sigma_2\theta'_2 = \theta_1[\mathcal{W}_1]$ by IH, then $X\sigma_1\sigma_2\theta'_2 \equiv X\sigma_2\theta'_2$ (as $X \notin dom(\sigma_1)$), $X\sigma_2\theta'_2 \equiv X\theta_1$ (as $X \in \mathcal{W}_1$), $X\theta_1 \equiv X\sigma_1\theta_1$ (as $X \notin dom(\sigma_1)$).
– Condition *iii)* : $(dom(\theta') \cup vRan(\theta')) \cap \mathcal{B} = \emptyset$: That is $(dom(\theta'_2) \cup vRan(\theta'_2)) \cap \mathcal{B} = \emptyset$, which happens as $(dom(\theta'_2) \cup vRan(\theta'_2)) \cap \mathcal{B}_1 = \emptyset$ by IH and $\mathcal{B} \subseteq \mathcal{B}_1$.

Proof (For theorem 8). Applying lemma 6 to $e\theta|_{FV(e)} \to^{l^*} e'$ with $\mathcal{W} = FV(e)$ and $\mathcal{B} = BV(e)$, as $e\theta|_{FV(e)} \equiv e\theta$.

5 A case of study: correctness of bubbling

Proof (For Theorem 9, Correctness of bubbling). The proof uses the following easy lemma about semantics of ?, which justifies also the equation $[\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!] = [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!]$ stated in the Theor. 9.

Lemma 17. $[\![e_1?e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$, *for any* $e_1, e_2 \in LExp_\perp$

Proof. We must prove $e_1?e_2 \twoheadrightarrow t \Leftrightarrow e_1 \twoheadrightarrow t \vee e_2 \twoheadrightarrow t$. Both implications are straigtforward using the rules of HOCRWL. For instance, for $\Rightarrow$, assume $e_1?e_2 \twoheadrightarrow t$. The derivation must use the HOCRWL rule (OR) and take the form

$$\frac{e_1 \twoheadrightarrow s \quad s \twoheadrightarrow t}{e_1?e_2 \twoheadrightarrow t}$$

if the rule $X?\ Y \to X$ of ? has been used in (OR), or a similar form with $e_2 \twoheadrightarrow s$ if $X?\ Y \to Y$ was used instead. But $e_1 \twoheadrightarrow s \quad s \twoheadrightarrow t$ implies $e_1 \twoheadrightarrow t$, and similar for e_2.

Coming back to the proof of Th. 9, we reason by induction on the number k of *let*'s occurring in $\mathcal{C}[e_1?e_2]$.

- $k = 0$: Since there is no *let* in $e_1?e_2$, we can apply Theor. 1 to obtain:

$$\begin{array}{ll}
[\![\mathcal{C}[e_1?e_2]]\!] & = \textit{(by Theor. 1)} \\
\bigcup_{t\in[\![e_1?e_2]\!]}[\![\mathcal{C}[t]]\!] & = \textit{(by Lemma 17)} \\
\bigcup_{t\in([\![\mathcal{C}[e_1]]\!]\cup[\![\mathcal{C}[e_2]]\!])}[\![\mathcal{C}[t]]\!] & = \textit{(set operations)} \\
\bigcup_{t\in[\![\mathcal{C}[e_1]]\!]}[\![\mathcal{C}[t]]\!] \cup \bigcup_{t\in[\![\mathcal{C}[e_2]]\!]}[\![\mathcal{C}[t]]\!] & = \textit{(by Theor. 1)} \\
[\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 17)} \\
[\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!] &
\end{array}$$

- $k > 0$: We reason by induction on the structure of $\mathcal{C}$.

– $\mathcal{C} \equiv [\ ]$: the result is trivial.
– $\mathcal{C} \equiv \mathcal{C}'\ e$: then

$$\begin{array}{ll}
[\![\mathcal{C}[e_1?e_2]]\!] & = \\
[\![\mathcal{C}'[e_1?e_2]\ e]\!] & = \textit{(by Theor. 2)} \\
\bigcup_{t\in[\![\mathcal{C}'[e_1?e_2]]\!]}[\![t\ e]\!] & = \textit{(by IH on } \mathcal{C}') \\
\bigcup_{t\in[\![\mathcal{C}'[e_1]?\mathcal{C}'[e_2]]\!]}[\![t\ e]\!] & = \textit{(by Lemma 17)} \\
\bigcup_{t\in([\![\mathcal{C}'[e_1]]\!]\cup[\![\mathcal{C}'[e_2]]\!])}[\![t\ e]\!] & = \textit{(set operations)} \\
\bigcup_{t\in[\![\mathcal{C}'[e_1]]\!]}[\![t\ e]\!] \cup \bigcup_{t\in[\![\mathcal{C}'[e_2]]\!]}[\![t\ e]\!] & = \textit{(by Theor. 2)} \\
[\![\mathcal{C}'[e_1]e]\!] \cup [\![\mathcal{C}'[e_2]e]\!] & = \\
[\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 17)} \\
[\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!] &
\end{array}$$

– $\mathcal{C} \equiv e\ \mathcal{C}'$: very similar to the previous one
– $\mathcal{C} \equiv \textit{let } x = e \textit{ in } \mathcal{C}'$: then

$$\begin{array}{ll}
[\![\mathcal{C}[e_1?e_2]]\!] & = \\
[\![\textit{let } x{=}e \textit{ in } \mathcal{C}'[e_1?e_2]]\!] & = \textit{(by Theor. 2,}\sigma \equiv \{x/t\}\textit{)} \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'[e_1?e_2]\sigma]\!] & = \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_1\sigma?e_2\sigma]]\!] & = \textit{(by IH on } k\textit{, that decreases)} \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_1\sigma]?\mathcal{C}'\sigma[e_2\sigma]]\!] & = \textit{(by Lemma 17)} \\
\bigcup_{t\in[\![e]\!]}([\![\mathcal{C}'\sigma[e_1\sigma]]\!] \cup [\![\mathcal{C}'\sigma[e_2\sigma]]\!]) & = \textit{(set operations)} \\
\bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_1\sigma]]\!] \cup \bigcup_{t\in[\![e]\!]}[\![\mathcal{C}'\sigma[e_2\sigma]]\!] & = \textit{(by Theor. 2)} \\
[\![\textit{let } x{=}e \textit{ in } \mathcal{C}'[e_1]]\!] \cup [\![\textit{let } x{=}e \textit{ in } \mathcal{C}'[e_2]]\!] & = \\
[\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!] & = \textit{(by Lemma 17)} \\
[\![\mathcal{C}[e_1]?\mathcal{C}[e_2]]\!] &
\end{array}$$

– $\mathcal{C} \equiv \textit{let } x = \mathcal{C}' \textit{ in } e$: very similar to the previous case

This ends the proof. It is interesting to observe that most of it consists of direct calculations with denotation of expressions, in the form of chains of equalities of denotations. We find this methodology quite appealing.

6 Translation to first order

Proof (For Proposition 2). Let us consider an expression $e = fo(e_{ho})$ (for some HO expression e_{ho}) and e' resulting from e by reducing one of its @-calls. We

want to prove $[\![e]\!] = [\![e']\!]$. The extension to consider any number of reductions of calls to @ in e is a straightforward induction on such a number.

The situation can be reflected by considering $e = C\ [@(h_n(e_1, \ldots, e_n), e_{n+1})]$ and two possible cases for e', depending on the two possible forms of the @-rule used:

- if $@(h_n(X_1, \ldots, X_n), Y) \to h_{n+1}(X_1, \ldots, X_n, Y) \in \mathcal{P}_{fo}$, with $h_n, h_{n+1} \in CS_{fo}$, then $e' = C\ [h_{n+1}(e_1, \ldots, e_n, e_{n+1})]$. It is trivial to prove that

$$[\![@(h_n(e_1, \ldots, e_n), e_{n+1})]\!] = [\![h_{n+1}(e_1, \ldots, e_n, e_{n+1})]\!]$$

  (because this @-rule is in fact the only applicable). Now, by Th. 1 we have

$$[\![e]\!] = [\![C\ [@(h_n(e_1, \ldots, e_n), e_{n+1})]]\!] = \bigcup_{t \in [\![@(h_n(e_1,\ldots,e_n),e_{n+1})]\!]} C\ [t] = \bigcup_{t \in [\![h_{n+1}(e_1,\ldots,e_n,e_{n+1})]\!]} C\ [t] = [\![C\ [h_{n+1}(e_1, \ldots, e_n, e_{n+1})]]\!] = [\![e']\!]$$

  .
- if $@(h_n(X_1, \ldots, X_n), Y) \to h(X_1, \ldots, X_n, Y) \in \mathcal{P}_{fo}$, with $h_n \in CS_{fo}$ and $h \in CS_{fo}$ (henceforth $h \in FS^n$), then $e' = C\ [h(e_1, \ldots, e_n, e_{n+1})]$. As before $[\![e]\!] = [\![e']\!]$, and by Th. 1 we obtain the result in a similar way to the previous case.

Lemma 18. *$(fo(e))[X/fo(t)] = fo(e[X/t])$.*

Proof. An easy induction over the structure of e.

Proof (For 10, Theorem Adequacy of HO-to-FO translation).

For the correctness of the transformation it is easier to use the alternative version of $HOCRWL_{let}$ of Fig. 4. Using this calculus we are in fact proving a generalized version of the result because it is proved for let-expressions instead of standard-expressions. So, the reformulation of the Theorem becomes:

Let $\mathcal{P}$ be a $HOCRWL_{let}$-program, $e \in LExp_\bot$, $t \in HOPat_\bot$ and $\mathcal{P}_{fo}$, $fo(e) \in LExp_{fo,\bot}$, $fo(t) \in CTerm_\bot$ the corresponding transformed $CRWL_{let}$-program, expression and pattern respectively. Then:

$$\mathcal{P} \vdash_{HOCRWL'_{let}} e \twoheadrightarrow t \Leftrightarrow \mathcal{P}_{fo} \vdash_{CRWL_{let}} fo(e) \twoheadrightarrow fo(t)\ \downarrow_@$$

We reason both implications separately:

($\Rightarrow$) We proceed by induction on the length l of the proof for $\mathcal{P} \vdash_{HOCRWL_{let}} e \twoheadrightarrow t$

$l = 0$ The cases **(B)**, **(RR)** and **(DC)** with $c \in DC^0$ are trivial.

$l \Rightarrow l + 1$ For the sake of simplicity and using the Prop. 2 when we write $e \twoheadrightarrow fo(t)$ we will understand $e \twoheadrightarrow fo(t)\ \downarrow_@$. We have the following cases:

(DC) the proof will have the form (we reason with only two arguments for simplicity, but the extension to more arguments is direct):

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2 \twoheadrightarrow t_2}{h\ e_1\ e_2 \twoheadrightarrow h\ t_1\ t_2} \qquad h \in \Sigma, \text{ if } h\ t_1\ t_2 \text{ is a partial pattern}$$

We have $fo(h\ e_1\ e_2) = @(@(h_0, fo(e_1)), fo(e_2))$ but, by proposition 2 we can work with the equivalent expression $h_2(fo(e_1), fo(e_2))$. On the other hand $fo(h\ t_1\ t_2) = h_2(fo(t_1), fo(t_2))$. In $CRWL_{let}$ we can build:

$$\dfrac{\dfrac{}{fo(e_1) \twoheadrightarrow fo(t_1)}\ i.h. \quad \dfrac{}{fo(e_2) \twoheadrightarrow fo(t_2)}\ i.h. \quad \dfrac{}{h_2(fo(t_1), fo(t_2)) \twoheadrightarrow h_2(fo(t_1), fo(t_2))}\ DC^*}{h_2(fo(e_1), fo(e_2)) \twoheadrightarrow h_2(fo(t_1), fo(t_2))}\ DC$$

(OR) Now we have a proof like:

$$\dfrac{e_1 \twoheadrightarrow t_1\ e_2 \twoheadrightarrow t_2\ \ r \twoheadrightarrow t}{f\ e_1\ e_2 \twoheadrightarrow t} \qquad \begin{array}{l}\text{if } t \text{ is a partial pattern} \\ (f\ t_1\ t_2 \to r) \in [\mathcal{P}]_\bot\end{array}$$

Again we work with $fo(f\ e_1\ e_2) = f(fo(e_1), fo(e_2))$ and can build the proof:

$$\dfrac{\dfrac{}{fo(e_1) \twoheadrightarrow fo(t_1)}\ i.h. \quad \dfrac{}{fo(e_2) \twoheadrightarrow fo(t_1)}\ i.h. \quad \dfrac{}{fo(r) \twoheadrightarrow fo(t)}\ i.h.}{f(fo(e_1), fo(e_2)) \twoheadrightarrow fo(t)}\ OR$$

that is performed using the instance $(f(fo(t_1), fo(t_2)) \to fo(r)) \in [\mathcal{P}_{fo}]_\bot$

(Let) The proof is:

$$\dfrac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t} \qquad \text{if } t \text{ is a partial pattern}$$

We have $fo(let\ X = e_1\ in\ e_2) = (let\ X = fo(e_1)\ in\ fo(e_2))$ and the proof:

$$\dfrac{\dfrac{}{fo(e_1) \twoheadrightarrow fo(t_1)}\ i.h. \quad \dfrac{}{fo(e_2)[X/fo(t_1)] \overset{Lem.18}{=} fo(e_2[X/t_1]) \twoheadrightarrow fo(t)}\ i.h.}{let\ X = fo(e_1)\ in\ fo(e_2) \twoheadrightarrow fo(t)}\ Let$$

(**Ap**) We have:

$$\dfrac{e_1 \twoheadrightarrow t_1 \quad e_2 \twoheadrightarrow t_2 \quad (t_1\ t_2) \twoheadrightarrow t}{(e_1\ e_2) \twoheadrightarrow t}$$

Notice that if $\mathcal{P} \vdash_{HOCRWL'_{let}} (t_1\ t_2) \twoheadrightarrow t$ by i.h. we also have $\mathcal{P}_{fo} \vdash_{CRWL_{let}} @(fo(t_1), fo(t_2)) \twoheadrightarrow fo(t)$. This proof must be done by **(OR)**, using a rule $(@(s_1, s_2) \to r) \in \mathcal{P}_{fo}$ and $\theta \in CSubst_\bot$ in the following way:

$$\dfrac{fo(t_1) \twoheadrightarrow s_1\theta \quad fo(t_2) \twoheadrightarrow s_2\theta \quad r\theta \twoheadrightarrow fo(t)}{@(fo(t_1), fo(t_2)) \twoheadrightarrow fo(t)}\ OR$$

On the other hand, by i.h. we have $fo(e_1) \twoheadrightarrow fo(t_1)$ and then, as $fo(t_1) \twoheadrightarrow s_1\theta$ we also have $fo(e_1) \twoheadrightarrow s_1\theta$, and similarly $fo(e_2) \twoheadrightarrow s_2\theta$. Now, we use these facts for building our proof in $CRWL_{let}$. First of all $fo(e_1\ e_2) = @(fo(e_1), fo(e_2))$ and the proof will have the form:

$$\frac{fo(e_1) \twoheadrightarrow s_1\theta \quad fo(e_2) \twoheadrightarrow s_2\theta \quad r\theta \twoheadrightarrow fo(t)}{@(fo(e_1), fo(e_2)) \twoheadrightarrow fo(t)}\ OR$$

Using the same function rule and the same c-susbtitution.

($\Leftarrow$) We proceed by induction on the length l of the proof for $\mathcal{P}_{fo} \vdash_{CRWL_{let}} fo(e) \twoheadrightarrow fo(t)$:

$l = 0$ The cases **(B)**, **(RR)** or **(DC)** with $c \in CS^0$ are trivial.

$l \Rightarrow l + 1$ The possible proofs with length greater than one are:

- **(DC)** and **(Let)** are easy applications of i.h.
- If the proof is done by **(OR)** it can have two forms depending on the function rule applied: it can belong to $\mathcal{P}_@$ or come from a rule of the original program $\mathcal{P}$:
  - For the first case the proof is:

$$\frac{fo(e_1) \twoheadrightarrow fo(t_1)\theta \quad fo(e_2) \twoheadrightarrow fo(t_2)\theta \quad fo(r)\theta \twoheadrightarrow fo(t)}{@(fo(e_1), fo(e_2)) \twoheadrightarrow fo(t)}$$

    using a rule $(@(fo(t_1), fo(t_2)) \to fo(r)) \in \mathcal{P}_{fo}$ and $\theta \in CSubst_\perp$. Then $(e_1\ e_2)$ is a partial application and $(t_1\ t_2)$ is a pattern and it is easy to build the proof $(e_1\ e_2) \twoheadrightarrow (t_1\ t_2)$ in $HOCRWL'_{let}$ using i.h. and DC.
  - For the second one we have

$$\frac{fo(e_1) \twoheadrightarrow fo(t_1)\theta \ldots fo(e_n) \twoheadrightarrow fo(t_n)\theta \quad fo(r)\theta \twoheadrightarrow fo(t)}{fo(f(e_1, \ldots, e_n)) \twoheadrightarrow fo(t)}$$

    taking $(fo(f(t_1, \ldots, t_n)) \to fo(r)) \in \mathcal{P}_{fo}$ and $\theta \in CSubst_\perp$. Then it must be $(f(t_1, \ldots, t_n) \to r) \in \mathcal{P}$ and we have:

$$\frac{\overline{e_1 \twoheadrightarrow t_1\theta}\ i.h. \quad \ldots \overline{e_n \twoheadrightarrow t_n\theta}\ i.h. \quad \overline{r\theta \twoheadrightarrow t}\ i.h.}{f(e_1, \ldots, e_n) \twoheadrightarrow t}$$

8.1.9 A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems (Extended version)

A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems*

(Extended version: revision February 2010)

Tech. Rep. SIC-10-08, 2008

Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
juanrh@fdi.ucm.es

Abstract. Formalisms involving some degree of nondeterminism are frequent in computer science. In particular, various programming or specification languages are based on term rewriting systems where confluence is not required. In this paper we examine three concrete possible semantics for non-determinism that can be assigned to those programs. Two of them –call-time choice and run-time choice– are quite well-known, while the third one –plural semantics– is investigated for the first time in the context of term rewriting based programming languages. We investigate some basic intrinsic properties of the semantics and establish some relationships between them: we show that the three semantics form a hierarchy in the sense of set inclusion, and we prove that call-time choice and plural semantics enjoy a remarkable compositionality property that fails for run-time choice; finally, we show how to express plural semantics within run-time choice by means of a program transformation, for which we prove its adequacy.

1 Introduction

Term rewriting systems (*TRS's*) [4] have a long tradition as a suitable basic formalism to address a wide range of tasks in computer science, in particular, many specification languages [5, 8], theorem provers [26, 25, 6] and programming languages are based on TRS's. For instance, the syntax and theory of TRS's was the basis of the first formulations of *functional logic programming* (*FLP*) [12] through the idea of narrowing [11]. On the other hand, non-determinism is an expressive feature that has been used for a long time in system specification (e.g., non-deterministic Turing machines or automata) or for programming (the constructions of McCarthy [22] and Dijkstra [7] are classical examples). One of the appeals of term rewriting is its elegant way to express non-determinism through the use of a non-confluent TRS, obtaining a clean and high level representation of complex systems. In the field of FLP, non-confluent TRS's are used as programs to support non-strict non-deterministic functions, which are one of the most distinctive features of the paradigm [10, 3]. Those TRS's follow the constructor discipline also, corresponding to a value-based semantic view, in which the purpose of computations is to produce values made of constructors.

Therefore non-confluent constructor-based TRS's can be used as a common syntactic framework for FLP and rewriting. The set of rewrite rules constitutes a program and so we also call them *program rules*. Nevertheless the behaviour of current implementations of FLP and rewriting differ substantially, because the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions, that were explored in [24]. There the different language variants that result after adding non-determinism to a basic functional language were expounded, structuring the comparison as a choice among different options over several dimensions: strict/non-strict functions, angelic/demonic/erratic non-deterministic choices and *singular/plural semantics* for parameter passing. In the present paper we assume non-strict angelic non-determinism, and we are concerned about the last dimension only. The key difference is that under a singular

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), Promesas-CAM (S-0505/TIC/0407) and FAST-STAMP (TIN2008-06622-C03-01/TIN).

semantics, in the substitutions used to instantiate the program rules for function application, the variables of the program rules should range over single objects of the set of values considered; in a plural semantics those range over sets of objects. This has been traditionally identified with the distinction between *call-time choice* and *run-time choice* [14] parameter passing mechanisms. Under call-time choice a value for each argument is computed before performing parameter passing, this corresponds to call-by-value in a strict setting and to call-by-need in a non-strict setting, in which a partial value instead of a total value is computed. On the other hand, run-time-choice corresponds to call-by-name, each argument is copied without any evaluation and so the different copies of any argument may evolve in different ways afterwards. Thus, traditionally it has been considered that call-time choice parameter passing inducts a singular semantics while run-time choice inducts a plural semantics.

Example 1. Consider the TRS $\mathcal{P} = \{f(c(X)) \to d(X,X), X\ ?\ Y \to X, X\ ?\ Y \to Y\}$. With call-time choice/singular semantics to compute a value for the term $f(c(0?1))$ we must first compute a (partial) value for $c(0?1)$, and then we may continue the computation with $f(c(0))$ or $f(c(1))$ which yield $d(0,0)$ or $d(1,1)$. Note that $d(0,1)$ and $d(1,0)$ are not correct values for $f(c(0?1))$ in that setting.
On the other hand with run-time choice/plural semantics to evaluate the term $f(c(0?1))$:

- Under the run-time choice point of view, the step $f(c(0?1)) \to d(0?1, 0?1)$ is sound, hence not only $d(0,0)$ and $d(1,1)$ but also $d(0,1)$ and $d(1,0)$ are valid values for $f(c(0?1))$.
- Under the plural semantics point of view, we consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0?1)$ in which every element matches the argument pattern $c(X)$. Therefore the set $\{0,1\}$ can be used for parameter passing obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$, which evaluation yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$ and $d(1,0)$.

In general, call-time choice/singular semantics produces less results than run-time choice/ plural semantics.

A standard formulation for call-time choice[1] in FLP is the *CRWL*[2] logic [9, 10], which is implemented by current FLP languages like Toy [18] or Curry [13]; traditional term rewriting may be considered the standard semantics for run-time choice[3], and is the basis for the semantics of languages like Maude [5], but has been rarely [1] thought as a valuable global alternative to call-time choice for the value-based view of FLP. However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal [16]. Nevertheless the use of an operational notion like term rewriting as the semantic basis of a FLP language can lead us to confusing situations, not very compatible with the value-based semantic view that we wanted for the constructor-based TRS's used in FLP.

Example 2. Starting with the TRS of Example 1 we want to evaluate the expression $f(c(0)\ ?\ c(1))$ with run-time choice/plural semantics:

- Under the run-time choice point of view, that is, using term rewriting, the evaluation of the subexpression $c(0)?c(1)$ is needed in order to get an expression that matches the left hand side $f(c(X))$. Hence the derivations $f(c(0)?c(1)) \to f(c(0)) \to d(0,0)$ and $f(c(0)?c(1)) \to f(c(1)) \to d(1,1)$ are sound and compute the values $d(0,0)$ and $d(1,1)$, but neither $d(0,1)$ nor $d(1,0)$ are correct values for $f(c(0)?c(1))$.
- Under the plural semantics point of view, we consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0)?c(1)$ in which every element matches the argument pattern $c(X)$. Therefore the set $\{0,1\}$ can be used for parameter passing obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$ that yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$ and $d(1,0)$.

Which of these is the more suitable perspective for FLP?

[1] In fact angelic non-strict call-time choice.
[2] **C**onstructor-based **R**e**W**riting **L**ogic.
[3] In fact angelic non-strict run-time choice.

This problem did not appear in [24] because no pattern matching was present, nor in [14] because only call-time choice was adopted (and this conflict does not appear between the call-time choice and the singular semantics views). Choosing the run-time choice perspective of term rewriting has some unpleasant consequences. First of all the expression $f(c(0?1))$ has more values than the expression $f(c(0)?c(1))$, even when the only difference between them is the subexpressions $c(0?1)$ and $c(0)?c(1)$, which have the same values both in call-time choice, run-time choice and plural semantics. This is pretty incompatible with the value-based semantic view we are looking for in FLP. And this has to do with the loss of some desirable properties, present in *CRWL*, when switching to run-time choice. We will see how plural semantics recovers those properties, which are very useful for reasoning about computations. Furthermore it allows natural encodings of some programs that need to do some collecting work, as we will see later (Example 5). Hence we claim that the plural semantics perspective is more suitable for a value-based programming language.

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about *CRWL* and term rewriting systems. In Section 3 we introduce $\pi CRWL$, a variation of *CRWL* to express plural semantics, and present some of its properties. In Section 4 we discuss about the different properties of these semantics and prove the inclusion chain $CRWL \subseteq$ rewriting $\subseteq \pi CRWL$, that constitutes a hierarchy of semantics for non-determinism. Section 5 recalls that no straight simulation of *CRWL* in term rewriting can be done by a program transformation, and vice versa, and shows a novel transformation to simulate $\pi CRWL$ using term rewriting. Finally Section 6 summarizes some conclusions and future work. Fully detailed proofs, including some auxiliary results, can be found in Appendix A.

2 Preliminaries

2.1 Constructor based term rewriting systems

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects. Given a set $\mathcal{A}$ we denote by $\mathcal{A}^*$ the set of finite sequences of elements of that set. For any sequence $a_1 \ldots a_n \in \mathcal{A}^*$ and function $f : \mathcal{A} \to \{true, false\}$, by $a_1 \ldots a_n \mid f$ we denote the sequence constructed taking in order every element from $a_1 \ldots a_n$ for which f holds.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing **values**. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ϵ for the identity (or empty) substitution. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some θ_i; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. Given $W \subseteq \mathcal{V}$ we write $\theta|_W$ for the restriction of θ to W, and $\theta|_{\setminus D}$ is a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$.

A *constructor-based term rewriting system* $\mathcal{P}$ (CS) is a set of c-rewrite rules of the form $f(\overline{t}) \to r$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in

$\bar{t}$. In the present work we restrict ourselves to CS's not containing *extra variables*, i.e., CS's for which $var(r) \subseteq var(f(\bar{t}))$ holds for any rewrite rule; the extension of this work to rules with extra variables is a subject of future work. We assume that every CS $\mathcal{P}$ contains the rules $\{X \; ? \; Y \to X, X \; ? \; Y \to Y, if\ true\ then\ X \to X\}$, defining the behaviour of $\_?\_ \in FS^2$, $if\_then\_ \in FS^2$, both used in mixfix mode, and that those are the only rules for that function symbols. For the sake of conciseness we will often omit these rules when presenting a CS.

Given a TRS $\mathcal{P}$, its associated *rewrite relation* $\to_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\sigma] \to_{\mathcal{P}} \mathcal{C}[r\sigma]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\sigma \in Subst$. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$ or denote it by $\mathcal{P} \vdash e \to e'$ and $\mathcal{P} \vdash e \to^* e'$.

2.2 The CRWL framework

In the *CRWL* framework [9, 10], programs are CS's, also called *CRWL-programs* (or simply 'programs') from now on. To deal with non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot$, $CTerm_\bot$, $Subst_\bot$, $CSubst_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus able to derive reduction statements of the form $e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, meaning informally that t is (or approximates to) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice. The *CRWL*-proof calculus is presented in Figure 1. Rule **B** (bottom) allows any expression to be undefined

$$\textbf{(RR)}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad \textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1, \ldots, e_n) \twoheadrightarrow c(t_1, \ldots, t_n)}\quad c \in CS^n$$

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(OR)}\ \frac{e_1 \twoheadrightarrow p_1\theta \ldots e_n \twoheadrightarrow p_n\theta \quad r\theta \twoheadrightarrow t}{f(e_1, \ldots, e_n) \twoheadrightarrow t}\quad \begin{array}{l} f(p_1, \ldots, p_n) \to r \in \mathcal{P} \\ \theta \in CSubst_\bot \end{array}$$

Fig. 1. Rules of *CRWL*

or not evaluated (non-strict semantics). Rule **OR** (outer reduction) expresses that to evaluate a function call we must choose a compatible program rule, perform parameter passing (by means of a $CSubst_\bot$ θ) and then reduce the right-hand side. The use of partial c-substitutions in OR is essential to express call-time choice, as only single partial values are used for parameter passing.

We write $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ to express that $e \twoheadrightarrow t$ is derivable in the *CRWL*-calculus using the program $\mathcal{P}$. Given a program $\mathcal{P}$, the *CRWL-denotation* of an expression $e \in Exp_\bot$ is defined as $[\![e]\!]^{sg}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$. In the following, we will usually omit the reference to $\mathcal{P}$.

3 π*CRWL*: a plural semantics for FLP

The new calculus π*CRWL* is defined by modifying the rules of *CRWL* to consider sets of partial values for parameter passing instead of single partial values: hence, only the rule OR should be modified. To avoid the need to extend the syntax with new constructions to represent those "set expressions" that we talked about in the introduction, we will exploit the fact that $[\![e_1 \; ? \; e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$. Therefore the substitutions used for parameter passing will map variables to "disjunctions of values". We define the set $CSubst^?_\bot = \{\theta \in Subst_\bot \mid \forall X \in dom(\theta), \theta(X) = t_1 \; ? \; \ldots \; ? \; t_n$ such that $t_1, \ldots, t_n \in CTerm_\bot, n > 0\}$, for which $CSubst_\bot \subseteq CSubst^?_\bot \subseteq Subst_\bot$ obviously holds. The operator $? : CSubst^*_\bot \to CSubst^?_\bot$ constructs the

$CSubst^?_\perp$ corresponding to a non empty sequence of $CSubst_\perp$, and is defined as $?(\theta_1 \dots \theta_n)(X) = X$ if $X \notin \bigcup_{i\in\{1,\dots,n\}} dom(\theta_i)$; $?(\theta_1 \dots \theta_n)(X) = \rho_1(X)\ ?\ \dots\ ?\ \rho_m(X)$, where $\rho_1 \dots \rho_m = \theta_1 \dots \theta_n \mid \lambda\theta.(X \in dom(\theta))$, otherwise. Then $dom(?(\theta_1 \dots \theta_n)) = \bigcup_i dom(\theta_i)$. This operator is overloaded to handle finite non empty sets $\Theta \subseteq CSubst_\perp$ as $?\Theta = ?(\theta_1 \dots \theta_n)$ where the sequence $\theta_1 \dots \theta_n$ corresponds to an arbitrary reordering of the elements of Θ.

The $\pi CRWL$-proof calculus is presented in Figure 2. The only difference with the calculus in Figure 1 is that the rule OR has been replaced by **POR** (plural outer reduction), in which we may compute more that one partial value for each argument, and then use a substitution from $CSubst^?_\perp$ instead of $CSubst_\perp$ for parameter passing, achieving a plural semantics[4]. This calculus derives reduction statements of the form $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ that express that t is (or approximates to) a possible value for e in this semantics, under the program $\mathcal{P}$. The $\pi CRWL$-*denotation* of an expression $e \in Exp_\perp$ under a program $\mathcal{P}$ in $\pi CRWL$ is defined as $[\![e]\!]^{pl}_{\mathcal{P}} = \{t \in CTerm_\perp \mid \mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t\}$.

$$\textbf{(RR)}\ \frac{}{X \twoheadrightarrow X}\quad X \in \mathcal{V} \qquad \textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1 \dots e_n \twoheadrightarrow t_n}{c(e_1,\dots,e_n) \twoheadrightarrow c(t_1,\dots,t_n)}\quad c \in CS^n$$

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \perp} \qquad \textbf{(POR)}\ \frac{\begin{array}{ccc} e_1 \twoheadrightarrow p_1\theta_{11} & & e_n \twoheadrightarrow p_n\theta_{n1} \\ \dots & \dots & \dots \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & e_n \twoheadrightarrow p_n\theta_{nm_n} \quad r\theta \twoheadrightarrow t \end{array}}{f(e_1,\dots,e_n) \twoheadrightarrow t}$$

$$(f(\overline{p}) \to r) \in \mathcal{P},\ \theta = ?\{\theta_{11},\dots,\theta_{1m_1}\} \uplus \dots \uplus ?\{\theta_{n1},\dots,\theta_{nm_n}\}$$
$$\forall i,j\ \theta_{ij} \in CSubst_\perp \wedge dom(\theta_{ij}) = var(p_i), \forall i\ m_i > 0$$

Fig. 2. Rules of $\pi CRWL$

Example 3. Consider the program of Example 1, that is $\mathcal{P} = \{f(c(X)) \to d(X,X), X\ ?\ Y \to X, X\ ?\ Y \to Y\}$. The following is a $\pi CRWL$-proof for the statement $f(c(0)?c(1)) \twoheadrightarrow d(0,1)$ (some steps have been omitted for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \twoheadrightarrow 0}\ DC}{c(0) \twoheadrightarrow c(0)}\ DC \quad \dfrac{}{c(1) \twoheadrightarrow \perp}\ B \quad c(0) \twoheadrightarrow c(0)}{c(0)?c(1) \twoheadrightarrow c(0)}\ POR \quad c(0)?c(1) \twoheadrightarrow c(1) \quad \dfrac{0?1 \twoheadrightarrow 0 \quad 0?1 \twoheadrightarrow 1}{d(0?1, 0?1) \twoheadrightarrow d(0,1)}\ DC}{f(c(0)?c(1)) \twoheadrightarrow d(0,1)}\ POR$$

$\pi CRWL$ enjoys some nice properties, like the following monotonicity property, where for any proof we define its *size* as the number of applications of rules of the calculus.

Lemma 1. *For any CRWL-program, $e, e' \in Exp_\perp$, $t, t' \in CTerm_\perp$ if $e \sqsubseteq e'$ and $t' \sqsubseteq t$ then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ implies $\mathcal{P} \vdash_{\pi CRWL} e' \twoheadrightarrow t'$ with a proof of the same size or smaller.*

One of the most important properties is its compositionality, a property very close to the DET-additivity property for algebraic specifications of [14]:

Theorem 1. *For any CRWL-program, $\mathcal{C} \in Contx$ and $e \in Exp_\perp$, $[\![\mathcal{C}[e]]\!]^{pl} = \bigcup_{\{t_1,\dots,t_n\} \subseteq [\![e]\!]^{pl}} [\![\mathcal{C}[t_1\ ?\ \dots\ ?\ t_n]]\!]^{pl}$, for any arrangement of the elements of $\{t_1,\dots,t_n\}$ in $t_1\ ?\ \dots\ ?\ t_n$.*

The proof for that theorem is based upon the commutativity, associativity of ?, and the idempotence of its partial application (see Appendix A).

$\pi CRWL$ also has some monotonicity properties related to substitutions. We define the preorder $\sqsubseteq_\pi$ over

[4] In fact angelic non-strict plural non-determinism.

$CSubst^?_\bot$ by $\theta \sqsubseteq_\pi \theta'$ iff $\forall X \in \mathcal{V}$, given $\theta(X) = t_1 \ ? \ \ldots \ ? \ t_n$ and $\theta'(X) = t'_1 \ ? \ \ldots \ ? \ t'_m$ then $\forall t \in \{t_1, \ldots, t_n\} \exists t' \in \{t'_1, \ldots, t'_m\}$ such that $t \sqsubseteq t'$; and the preorder $\trianglelefteq$ over $Subst_\bot$ by $\sigma \trianglelefteq \sigma'$ iff $\forall X \in \mathcal{V}$, $[\![\sigma(X)]\!]^{pl} \subseteq [\![\sigma'(X)]\!]^{pl}$.

Lemma 2. *For any CRWL-program, $e \in Exp_\bot$, $t \in CTerm_\bot$, $\sigma, \sigma' \in Subst_\bot$, $\theta, \theta' \in CSubst^?_\bot$:*

1. ***Strong monotonicity of $Subst_\bot$**: If $\forall X \in \mathcal{V}, s \in CTerm_\bot$ given $\mathcal{P} \vdash_{\pi CRWL} \sigma(X) \twoheadrightarrow s$ with size K we also have $\mathcal{P} \vdash_{\pi CRWL} \sigma'(X) \twoheadrightarrow s$ with size $K' \leq K$, then $\vdash_{\pi CRWL} e\sigma \twoheadrightarrow t$ with size L implies $\vdash_{\pi CRWL} e\sigma' \twoheadrightarrow t$ with size $L' \leq L$.*
2. ***Monotonicity of $CSubst_\bot$**: If $\theta, \theta' \in CSubst_\bot$ and $\theta \sqsubseteq \theta'$ then $\mathcal{P} \vdash_{\pi CRWL} e\theta \twoheadrightarrow t$ with size K implies $\mathcal{P} \vdash_{\pi CRWL} e\theta' \twoheadrightarrow t$ with size $K' \leq K$.*
3. ***Monotonicity of $Subst_\bot$**: If $\sigma \trianglelefteq \sigma'$ then $[\![e\sigma]\!]^{pl} \subseteq [\![e\sigma']\!]^{pl}$.*
4. ***Monotonicity of $CSubst^?_\bot$**: If $\theta \sqsubseteq_\pi \theta'$ then $[\![e\theta]\!]^{pl} \subseteq [\![e\theta']\!]^{pl}$.*

Until now we have seen that $\pi CRWL$ enjoys the fundamental properties of *CRWL*, namely compositionality and monotonicity for expressions and substitutions. Nevertheless, there are some properties of *CRWL*—and as a consequence, of call-time choice—that do not hold for $\pi CRWL$. One of these is the correctness of the *bubbling* operational rule [2], which can be formulated as "for any *CRWL*-program, $\mathcal{C} \in Contx$ and $e_1, e_2 \in Exp_\bot$, $[\![\mathcal{C}[e_1 \ ? \ e_2]]\!] = [\![\mathcal{C}[e_1] \ ? \ \mathcal{C}[e_2]]\!]$". Note that examples 1 and 2 already show that this property does not hold for run-time choice, the following (counter)example proves that it is not the case for $\pi CRWL$ neither[5].

Example 4. Consider the program $\mathcal{P} = \{pair(X) \to (X, X), X \ ? \ Y \to X, X \ ? \ Y \to Y\}$ and the expressions $pair(0 \ ? \ 1)$ and $pair(0) \ ? \ pair(1)$ which correspond to a bubbling step using $\mathcal{C} = pair([])$. It is easy to check that $(0, 1) \in [\![pair(0 \ ? \ 1)]\!]^{pl}$ while $(0, 1) \notin [\![pair(0) \ ? \ pair(1)]\!]^{pl}$.

We end this section with an example of the use of $\pi CRWL$ to model problems in which some collecting work has to be done.

Example 5. We want to represent the database of a bank in which we hold some data about its employees, this bank has several branches and we want to organize the information according to them. So we define a non-deterministic function *branches* to represent the set of branches: a set is identified then with a non-deterministic expression. In this line we define a non-deterministic function *employees* which conceptually returns the set of records containing the information regarding the employees that work in a branch. Now, to search for the names of two clerks we define the function *twoclerks* which is based upon $find$, which forces the desired pattern $e(N, S, clerk)$ over the set defined by $employees(branches)$.

$\mathcal{P} = \{branches \to madrid, branches \to vigo, employees(madrid) \to e(pepe, men, clerk), employees(madrid) \to e(paco, men, boss), employees(vigo) \to e(maria, women, clerk), employees(vigo) \to e(jaime, women, boss), twoclerks \to find(employees(branches)), find(e(N, S, clerk)) \to (N, N)\}$

With term rewriting $twoclerks \to find(employees(branches)) \not\twoheadrightarrow^* (pepe, maria)$, because in that expression the evaluation of *branches* is needed and so one of the branches must be chosen. On the other hand with $\pi CRWL$ (some steps have been omitted for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk)}\ POR \quad \dfrac{\dots}{(pepe \ ? \ maria, pepe \ ? \ maria) \twoheadrightarrow (pepe, maria)}\ DC}{(pepe \ ? \ maria, pepe \ ? \ maria) \twoheadrightarrow (pepe, maria)}\ POR}{employees(branches) \twoheadrightarrow e(maria, \bot, clerk)}\ POR}{find(employees(branches)) \twoheadrightarrow (pepe, maria)}\ POR}{twoclerks \twoheadrightarrow (pepe, maria)}\ POR$$

where

$$\dfrac{\dfrac{}{branches \twoheadrightarrow madrid}\ POR \quad \dfrac{\dots}{e(pepe, men, clerk) \twoheadrightarrow e(pepe, \bot, clerk)}\ DC}{employees(branches) \twoheadrightarrow e(pepe, \bot, clerk)}\ POR$$

[5] In the originally published short version of this paper [23] the correctness of bubbling for $\pi CRWL$ was stated as Theorem 6. Although this is an important erratum, its impact in the rest of the work is negligible, as it is not used in the proof of any of the other results in the paper.

4 Comparison: a hierarchy of semantics

When comparing these semantics is not surprising finding that *CRWL* and π*CRWL* have similar properties. For example the monotonicity Lemma 1 also holds for *CRWL*; this lemma does not even make sense for term rewriting, as it only works with total terms. On the other hand term rewriting is closed under $Subst$ ($e \to^* e'$ implies $e\sigma \to^* e'\sigma$, for any $\sigma \in Subst$); *CRWL* is not closed under $Subst$ but under $CSubst_\bot$, as corresponds to call-time choice; π*CRWL* is closed under $CSubst_\bot$ too (see Appendix A), and we conjecture that for $\theta \in CSubst^?_\bot$ if $\vdash_{\pi CRWL} e \twoheadrightarrow t$ then $[\![t\theta]\!]^{pl} \subseteq [\![e\theta]\!]^{pl}$. For *CRWL* a compositionality result similar to Theorem 1 also holds, and bubbling is correct too [17]. This is not the case for term rewriting, as we saw when switching from $f(c(0?1))$ to $f(c(0)?c(1))$ in examples 1 and 2.

4.1 The hierarchy

As π*CRWL* is a modification of *CRWL*, the relation between them is very direct.

Theorem 2. *For any CRWL-program $\mathcal{P}$, $e \in Exp_\bot, t \in CTerm_\bot$ given a CRWL-proof for $\mathcal{P} \vdash e \twoheadrightarrow t$ we can build a πCRWL-proof for $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ just replacing every **OR** step by the corresponding **POR** step. As a consequence $[\![e]\!]^{sg}_{\mathcal{P}} \subseteq [\![e]\!]^{pl}_{\mathcal{P}}$.*

Concerning the relation of *CRWL* and π*CRWL* with term rewriting, we will use the notion of *shell* $|e|$ *of an expression* e that represents the outer constructor (and thus computed) part of e, defined as $|\bot| = \bot$, $|X| = X$, $c(e_1, \ldots, e_n) = c(|e_1|, \ldots, |e_n|)$, $|f(e_1, \ldots, e_n)| = \bot$ (for $X \in \mathcal{V}, c \in CS, f \in FS$). We also define the denotation of $e \in Exp$ under term rewriting as $[\![e]\!]^{rw} = \{t \in CTerm_\bot \mid \exists e' \in Exp \,.\, e \to^* e' \wedge t \sqsubseteq |e'|\}$. In a previous joint work the author explored the relation between *CRWL* and term rewriting ([15], Theorem 9), recast in the following theorem:

Theorem 3. *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rw}$. The converse inclusion does not hold in general.*

As we saw in Example 1, in general call-time choice semantics like *CRWL* produce less results than run-time choice semantics like the one induced by term rewriting. We will see that this kind of relation also holds for term rewriting and π*CRWL*.

Theorem 4. *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{rw} \subseteq [\![e]\!]^{pl}$. The converse inclusion does not hold in general.*

The key for proving Theorem 4 is a lemma stating that $\forall e, e' \in Exp$ if $e \to e'$ then $[\![e']\!]^{pl} \subseteq [\![e]\!]^{pl}$, that is, that every rewriting step is sound wrt. π*CRWL*. The evident corollary for these theorems is the announced inclusion chain.

Corollary 1. *For any CRWL-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rw} \subseteq [\![e]\!]^{pl}$. Hence $\forall t \in CTerm$, $\vdash_{CRWL} e \twoheadrightarrow t$ implies $e \to^* t$ which implies $\vdash_{\pi CRWL} e \twoheadrightarrow t$.*

5 Simulating plural semantics

In [15, 16] it was shown that neither *CRWL* can be simulated by term rewriting with a simple program transformation, nor vice versa. Nevertheless, plural semantics can be simulated by rewriting using the transformation presented in the current section, which could be used as the basis for a first implementation of π*CRWL* that we might use for experimentation. First we will present a naive version of this transformation, and show its adequacy; later we will propose some simple optimizations for it.

5.1 A simple transformation

Definition 1. *Given a CRWL-program P, for every rule* $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ *such that* $f \notin \{\_?\_, if\_then\_\}$ *we define its transformation as:*

$pST(f(p_1,\ldots,p_n) \to r) = f(Y_1,\ldots,Y_n) \to if\ match(Y_1,\ldots,Y_n)\ then\ r[\overline{X_{ij}/project_{ij}(Y_i)}]$

- $\forall i \in \{1,\ldots,n\}$, $\{X_{i1},\ldots,X_{ik_i}\} = var(p_i) \cap var(r)$ *and* $Y_i \in \mathcal{V}$ *is fresh.*
- $match \in FS^n$ *fresh is defined by the rule* $match(p_1,\ldots,p_n) \to true$.
- *Each* $project_{ij} \in FS^1$ *is a fresh symbol defined by the single rule* $project_{ij}(p_i) \to X_{ij}$.

For $f \in \{\_?\_, if\_then\_\}$ *the transformation leaves its rules untouched.*

The function *match* is used to impose a "guard" that enforces the matching of each argument with its corresponding pattern. If we dropped this condition the translation of, for example, to rule $(null(nil) \to true)$, would be $(null(Y) \to true)$, which is clearly unsound as then $null(0 : nil) \to true$. Besides each pattern p_i has been replaced by a fresh variable Y_i, to which any expression can match, hence the arguments may be replicated freely by the rewriting process without demanding any evaluation and thus keeping its denotation untouched: this is the key to achieve completeness wrt. $\pi CRWL$. Later on, the functions $project_{ij}$ will just make the projection of each variable when needed.

Example 6. Applying this to Example 1 we get
$\{f(Y) \to if\ match(Y)\ then\ d(project(Y), project(Y)), match(c(X)) \to true, project(c(X)) \to X\}$
under which we can do:
$\underline{f(c(0)?c(1))} \to if\ \underline{match(c(0)?c(1))}\ then\ d(project(c(0)?c(1)), project(c(0)?c(1)))$
$\to^* if\ true\ then\ \overline{d(project(c(0)?c(1)), project(c(0)?c(1)))}$
$\to \overline{d(project(\underline{c(0)?c(1)}), project(\underline{c(0)?c(1)}))} \to^* d(\underline{project(c(0))}, \underline{project(c(1))}) \to^* d(0,1)$

Concerning the adequacy of the transformation:

Theorem 5. *For any CRWL-program* $\mathcal{P}$, $e \in Exp_\perp$ *built up on the signature of* $\mathcal{P}$, $[\![e]\!]^{pl}_{pST(\mathcal{P})} \subseteq [\![e]\!]^{pl}_{\mathcal{P}}$.

Theorem 6. *For any CRWL-program* $\mathcal{P}$, $e \in Exp, t \in CTerm_\perp$ *built up on the signature of* $\mathcal{P}$, *if* $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ *then exists some* $e' \in Exp$ *built using symbols of the signature of* $pST(\mathcal{P})$ *such that* $pST(\mathcal{P}) \vdash e \to^* e'$ *and* $t \sqsubseteq |e'|$.

Corollary 2. *For any CRWL-program* $\mathcal{P}$, $e \in Exp$ *built using symbols of the signature of* $\mathcal{P}$, $[\![e]\!]^{pl}_{\mathcal{P}} = [\![e]\!]^{rw}_{pST(\mathcal{P})}$. *Hence* $\forall t \in CTerm\ \mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ *iff* $pST(\mathcal{P}) \vdash e \to^* t$.

5.2 An optimized transformation

Concerning the transformation, if a pattern is ground then no parameter passing will be done for it and so no transformation is needed: for $null(nil) \to true$ we get $\{null(Y) \to if\ match(Y)\ then\ true, match(nil) \to true\}$, which is equivalent. Besides, if the pattern is a variable then any expression matches it and the projection functions are trivial, so no transformation is needed neither, as happens with $pair(X) \to (X, X)$ for which $\{pair(Y) \to if\ match(Y)\ then\ (project(Y), project(Y)), match(X) \to true, project(X) \to X\}$ are returned.

Definition 2. *Given a CRWL-program P, for every rule* $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ *we define its transformation as:*

$pST(f(p_1,\ldots,p_n) \to r)$

$$= \begin{cases} f(p_1,\ldots,p_n) \to r & \text{if } \rho_1\ldots\rho_m \text{ is empty} \\ f(\tau(p_1),\ldots,\tau(p_n)) \to \begin{array}{l} if\ match(Y_1,\ldots,Y_m) \\ \quad then\ r[\overline{X_{ij}/project_{ij}(Y_i)}] \end{array} & \text{otherwise} \end{cases}$$

where $\rho_1\ldots\rho_m = p_1\ldots p_n \mid \lambda p.(p \notin \mathcal{V} \wedge var(p) \neq \emptyset)$.

- $\forall \rho_i$, $\{X_{i1}, \ldots, X_{ik_i}\} = var(\rho_i) \cap var(r)$ *and* $Y_i \in \mathcal{V}$ *is fresh.*
- $\tau : CTerm \to CTerm$ *is defined by* $\tau(p) = p$ *if* $p \in \mathcal{V} \vee var(p) = \emptyset$ *and* $\tau(p) = Y_i$ *otherwise, for* $p \equiv \rho_i$.
- $match \in FS^m$ *fresh is defined by the rule* $match(\rho_1, \ldots, \rho_m) \to true$.
- *Each* $project_{ij} \in FS^1$ *is a fresh symbol defined by the single rule* $project_{ij}(\rho_i) \to X_{ij}$.

We will not give a formal proof for the adequacy of the optimization. Nevertheless note how this transformation leaves untouched the rules for $\_?\_$ and $if\_then\_$ without defining an special case for them. As the simple transformation worked well for that rules that suggests that we are doing the right thing. We end this section with an example application of the optimized transformation, over the program of Example 5:

Example 7. The only rule modified is the one for $find$, for which we get $\{find(Y) \to if\ match(Y)\ then\ (project(Y), project(Y)), match(e(N, s, clerk)) \to true, project(e(N, s, clerk)) \to N\}$ so:

$\underline{twoclerks} \to \underline{find(employees(branches))}$
$\to if\ match(\underline{employees(branches)})\ then\ (project(employees(branches)), project(employees(branches)))$
$\to^* \underline{if\ match(e(pepe, men, clerk))\ then\ (project(employees(branches)), project(employees(branches)))}$
$\to^* (project(\underline{employees(branches)}), project(\underline{employees(branches)}))$
$\to^* (\underline{project(e(pepe, men, clerk))}, \underline{project(e(maria, women, clerk))}) \to^* (pepe, maria)$

6 Conclusions

In this work we have pointed the different interpretations of run-time choice and plural semantics caused by pattern matching. To the best of our knowledge this distinction is stablished in the present paper for the first time, because in [24] no pattern matching was present and in [14] only call-time choice was adopted. We argue that the run-time choice semantics induced by term rewriting is not the best option for a value-based programming language like current implementations of FLP. For that context a plural semantics has been proposed for which the compositionality properties lost when turning from call-time choice to rewriting are recovered. Nevertheless, for other kind of rewriting based languages like Maude, which are not limited to constructor-based TRS's, term rewriting has been proven to be an effective formalism.

Our concrete contributions can be summarized as follows:
- We have presented the proof calculus $\pi CRWL$, a novel formulation of plural semantics for left-linear constructor-based TRS's, which are the kind of TRS's used in FLP. Some basic properties of the new semantics have been stated and proved, and by some examples we have shown how it allows natural encodings of some programs that need to do some collecting work (Sect. 3).
- We have compared the new calculus with $CRWL$ and term rewriting, which are standard formulations for call-time choice and run-time choice respectively. The different properties of these calculi have been discussed and the inclusion chain $CRWL \subseteq$ rewriting $\subseteq \pi CRWL$ has been proved (Sect. 4).
- We have recalled some previous results about the impossibility of a straight simulation of $CRWL$ in term rewriting or viceversa by a simple program transformation. Besides we have proposed a novel program transformation to simulate plural semantics with term rewriting, and proved its adequacy (Sect. 5).

From a practical point of view, it might be unrealistic to think that a monolithic semantic view is adequate for addressing all non-determinism present in a large program. In [16] we have started to investigate the combination of call-time choice and run-time choice in a unified framework. But as $\pi CRWL$ seems to be more suitable than run-time choice for a value-based language, we are planning to extend that work to plural semantics.

We contemplate other relevant subjects of future work:
- Extending the current results to programs with extra variables, that is, with rules $l \to r$ in which $var(r) \subseteq var(l)$ does not hold in general. We should also deal with conditional rules and equality constraints to cover the basic features of FLP languages.
- Studying the relation between the determinism of programs under $CRWL$ [15] and $\pi CRWL$, which we conjecture is equivalent. We also conjecture that for deterministic programs $\forall e \in Exp, [\![e]\!]^{sg} = [\![e]\!]^{rw} = [\![e]\!]^{pl}$. Getting results about the relation of confluence and determinism of programs could be useful for analyzing the confluence of a TRS through its determinism. In the same line, the inclusion chain $CRWL \subseteq$ rewriting

$\subseteq \pi CRWL$ could be used to study the termination of a TRS through its termination in $CRWL$ and $\pi CRWL$.
- Developing a more operational rewrite notion for $\pi CRWL$ in the line of [15], which could be extended to narrowing like in [17]. A complexity study would be needed to ensure that the extra nondeterminism does not preclude the design of an efficient implementation. On the other hand the natural value for $\pi CRWL$ seems to be $\mathcal{P}(CTerm_\bot)$ instead of $CTerm_\bot$, a formulation in the line of [19] could be useful to forget about the tricky use of $\_?\_$.
- Finally, for the immediate future, it could be interesting implementing the transformation to simulate $\pi CRWL$ in some term rewriting based language like Maude [5]. Maybe the context-sensitive rewriting [21] features of Maude could be used to improve the laziness of the transformed program like in [20]. Besides, the matching-module capacities of Maude could be used to enhance the expressivity of plural semantics.

Acknowledgements: The author would like to thank Paco López Fraguas and Jaime Sánchez Hernández for their support and their useful suggestions. I would also like to thank the referees for their very valuable comments.

References

1. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. ALP'97*, pages 16–30. Springer LNCS 1298, 1997.
2. S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In Proc. Termgraph'06, pages 61–70. ENTCS, 176(1), 2007.
3. S. Antoy and M. Hanus. Functional logic design patterns. In Proc. FLOPS'02, pages 67–87. Springer LNCS 2441, 2002.
4. F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University Press, United Kingdom, 1998.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Springer LNCS 4350, 2007
6. M. Clavel, M. Palomino, and A. Riesco. Introducing the itp tool: a tutorial. J. UCS 12(11), pages 1618–1650, 2006.
7. E. W. Dijkstra. A Discipline of Programming. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
8. R. Diaconescu and K. Futatsugi. An overview of CafeOBJ. ENTCS 15,1998.
9. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In Proc. ESOP'96, pages 156–172. Springer LNCS 1058, 1996.
10. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. J. Log. Program. 40(1), pages 47–87, 1999.
11. M. Hanus. The integration of functions into logic programming: From theory to practice. J. Log. Program. 19/20, pages 583–628, 1994.
12. M. Hanus. Multi-paradigm declarative languages. In Proc. ICLP'07, pages 45–75. Springer LNCS 4670, 2007.
13. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
14. H. Hussmann. Non-Determinism in Algebraic Specifications and Algebraic Programs. Birkhäuser Verlag, 1993.
15. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In Proc. PPDP'07, pages 197–208. ACM Press, 2007.
16. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions (Extended version). Tech. Rep. SIC-9-08, Universidad Complutense de Madrid, 2008. `http://gpd.sip.ucm.es/juanrh/pubs/tchrRTCT08.pdf`.
17. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In Proc. FLOPS'08, pages 147–162. Springer LNCS 4989, 2008.
18. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In Proc. RTA'99, pages 244–247. Springer LNCS 1631, 1999.
19. F. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. TPLP, 4(1&2), pages 41–74, 2004.
20. S. Lucas. Needed reductions with context-sensitive rewriting. In Proc. ALP/HOA'97, pages 129–143. Springer LNCS 1298, 1997.
21. S. Lucas. Context-sensitive computations in functional and functional logic programs. J. Fun. Log. Program 1998(1), 1998.

22. J. McCarthy. A Basis for a Mathematical Theory of Computation. Computer Programming and Formal Systems, pages 33–70. North-Holland, Amsterdam, 1963.
23. Juan Rodríguez-Hortalá. A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems. In Proc. FSTTCS'08, pages 328–339. LIPIcs 2, Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2008.
24. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. The Computer Journal 35(5), pages 514–523, 1992.
25. The Coq Development Team, LogiCal Project. The coq proof assistant reference manual version 8.1. Technical report, INRIA, 2006. `http://pauillac.inria.fr/coq/V8.1pl3/refman`.
26. M. Wenzel. The isabelle/isar reference manual. `http://isabelle.in.tum.de/dist/Isabelle99-2/doc/isar-ref.pdf`.

A Proofs of the results

During the proofs we will often use the notation *IH* to refer to the induction hypothesis. We will also use the following notions:

Definition 3.

- *The set of positions of an expression e is the set $\mathcal{O}(e)$ of strings over the alfabet of positive integers defined as $\mathcal{O}(X) = \{\epsilon\}$, if $X \in \mathcal{V}$; $\mathcal{O}(h(e_1, \ldots, e_n)) = \{\epsilon\} \cup \bigcup_{i \in \{1,\ldots,n\}} \{i.p \mid p \in \mathcal{O}(e_i)\}$ otherwise, where ϵ denotes the empty string and $\_.\_$ is used for concatenation.*
  We say that two positions are parallel if none of them is prefix of the other.
- *For any $e \in Exp_\bot$, $p \in \mathcal{O}(()e)$, the subexpression of e at position p denoted by $s|_p$, is defined as $e|_\epsilon = e$; $h(e_1, \ldots, e_n)|_{i.q} = e_i|_q$.*
- *For any $e, e' \in Exp_\bot$, $p \in \mathcal{O}(e)$, by $e[e']_p$ we denote the expression obtained from e by replacing the subexpression at position p by e', defined as $e[e']_\epsilon = e'$; $f(e_1, \ldots, e_n)[e']_{i.q} = f(e_1, \ldots, e_i[e']_q, \ldots, e_n)$.*
- *As one-hole context can be understood as funcions $\mathcal{C} : Exp_\bot \to Exp_\bot$, for any $\mathcal{C} \in Cntx$ we may assume some $e \in Exp_\bot, p \in \mathcal{O}(e)$ such that $\mathcal{C} = \lambda e'.e[e']_p$. With this in mind Theorem 1 can be recasted as $[\![e[e']_p]\!]^{pl} = \bigcup_{\bar{t} \subseteq [\![e']\!]^{pl}} [\![e[?\bar{t}]_p]\!]^{pl}$, where $?\{t_1, \ldots, t_n\}$ denotes $t_1 \ ? \ \ldots \ ? \ t_n$ for some arrangement of the elements of $\{t_1, \ldots, t_n\}$ in $t_1 \ ? \ \ldots \ ? \ t_n$.*

A.1 For Section 3

The following auxiliary lemmas will be used in the proofs:

Lemma 3. *For any $\pi CRWL$-program, $t, t' \in CTerm_\bot$, $e \in Exp_\bot$, $\sigma, \sigma' \in Subst_\bot$*

1. *$\mathcal{P} \vdash_{\pi CRWL} t \twoheadrightarrow t$*
2. *$\mathcal{P} \vdash_{\pi CRWL} t \twoheadrightarrow t'$ iff $t' \sqsubseteq t$.*
3. *If $\sigma \sqsubseteq \sigma'$ then $e\sigma \sqsubseteq e\sigma'$.*

Proof.

1. By a simple induction on the structure of t
2. Assume $\mathcal{P} \vdash_{\pi CRWL} t \twoheadrightarrow t'$, we can prove $t' \sqsubseteq t$ by a simple induction on the structure of t. For the converse implication assume $t' \sqsubseteq t$, then $t \twoheadrightarrow t$ by the previuos item, hence $t \twoheadrightarrow t'$ by Lemma 1.
3. A simple induction on the structure of e.

Proof (For Lemma 1). By induction on the structure of $e \twoheadrightarrow t$.

Base cases

B $e \twoheadrightarrow \bot \equiv t$. Then $t' \sqsubseteq t$ implies $t' \equiv \bot$, so $e' \twoheadrightarrow \bot \equiv t'$, by B.

RR $e \equiv X \twoheadrightarrow X \equiv t$. Then $t' \sqsubseteq t$ implies $t' \equiv \bot$ or $t' \equiv X$. In the first case we proceed like in the case for B, in the latter as $X \equiv e \sqsubseteq e'$ implies $e' \equiv X$ then $e' \equiv X \twoheadrightarrow X \equiv t'$ by RR.

DC $e \equiv c \twoheadrightarrow c \equiv t$. We can proceed in similar way we did in the previous case.

Inductive steps

DC Then we have

$$\frac{e_1 \twoheadrightarrow t_1 \ \ldots \ e_n \twoheadrightarrow t_n}{e \equiv c(e_1, \ldots, e_n) \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t} \ DC$$

Then $t' \sqsubseteq t$ implies $t' \equiv \bot$ or $t' \equiv c(t'_1, \ldots, t'_n)$ with $t'_i \sqsubseteq t_i$ for every $i \in \{1, \ldots, n\}$. In the first case we proceed like in the case for B, in the latter as $c(e_1, \ldots, e_n) \equiv e \sqsubseteq e'$ implies $e' \equiv c(e'_1, \ldots, e'_n)$ with $e_i \sqsubseteq e'_i$ for every $i \in \{1, \ldots, n\}$ then by IH $e'_i \twoheadrightarrow t'_i$ for every $i \in \{1, \ldots, n\}$, and we can build the following proof:

$$\frac{e'_1 \twoheadrightarrow t_1 \ \ldots \ e'_n \twoheadrightarrow t_n}{e' \equiv c(e'_1, \ldots, e'_n) \twoheadrightarrow c(t'_1, \ldots, t'_n) \equiv t'} \ DC$$

POR Then we have

$$\frac{\begin{array}{ccccc} e_1 \twoheadrightarrow p_1\theta_{11} & & & & e_n \twoheadrightarrow p_n\theta_{n1} \\ \dots & & \dots & & \dots \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & & & e_n \twoheadrightarrow p_n\theta_{nm_n} \qquad r\theta \twoheadrightarrow t \end{array}}{e \equiv f(e_1,\dots,e_n) \twoheadrightarrow t}\ POR$$

with $\theta = ?(\theta_{11}\dots\theta_{1m_1}) \uplus \dots \uplus ?(\theta_{n1}\dots\theta_{nm_n})$, for some $(f(p_1,\dots,p_n) \to r) \in \mathcal{P}$. Then as $f(e_1,\dots,e_n) \equiv e \sqsubseteq e'$ implies $e' \equiv f(e'_1,\dots,e'_n)$ with $e_i \sqsubseteq e'_i$ for every $i \in \{1,\dots,n\}$ then by IH $\forall i \in \{1,\dots,n\}, \forall j \in \{1,\dots,m_i\}$ $e'_i \twoheadrightarrow p_i\theta_{ij}$. We can also apply the IH to get $r\theta \twoheadrightarrow t'$, as $t' \sqsubseteq t$ by hypothesis, and build the following proof:

$$\frac{\begin{array}{ccccc} e'_1 \twoheadrightarrow p_1\theta_{11} & & & & e'_n \twoheadrightarrow p_n\theta_{n1} \\ \dots & & \dots & & \dots \\ e'_1 \twoheadrightarrow p_1\theta_{1m_1} & & & & e'_n \twoheadrightarrow p_n\theta_{nm_n} \qquad r\theta \twoheadrightarrow t' \end{array}}{e' \equiv f(e'_1,\dots,e'_n) \twoheadrightarrow t'}\ POR$$

Lemma 4. *For any CRWL-program, $\mathcal{C} \in Contx$ and $e_1, e_2, e_3 \in Exp_\perp$:*

1. $[\![\mathcal{C}[e1\ ?\ e_2]]\!]^{pl} = [\![\mathcal{C}[e_2\ ?\ e_1]]\!]^{pl}$
2. $[\![\mathcal{C}[(e1\ ?\ e_2)\ ?\ e_3]]\!]^{pl} = [\![\mathcal{C}[e_1\ ?\ (e_2\ ?\ e_3)]]\!]^{pl}$
3. $[\![\mathcal{C}[e_1\ ?\ e_1]]\!]^{pl} = [\![\mathcal{C}[e_1]]\!]^{pl}$
4. $[\![\mathcal{C}[e_1]]\!]^{pl} \subseteq [\![\mathcal{C}[e_1\ ?\ e_2]]\!]^{pl}$. *As a consequence, for any pair of finite chains $a_1\dots a_n \in Exp^*_\perp$, $b_1\dots b_m \in Exp^*_\perp$ if $\{a_1,\dots,a_n\} \subseteq \{b_1,\dots,b_m\}$ then for any context $\mathcal{C}$, $[\![\mathcal{C}[a_1\ ?\ \dots\ ?\ a_n]]\!]^{pl} \subseteq [\![\mathcal{C}[b_1\ ?\ \dots\ ?\ b_m]]\!]^{pl}$ holds.*

Proof (For Lemma 4 (Sketch)).

1. We have to prove that for any $t \in CTerm_\perp$ if $\mathcal{C}[e_1\ ?\ e_2] \twoheadrightarrow t$ then $\mathcal{C}[e_2\ ?\ e_1] \twoheadrightarrow t$ and vice versa. This can be easily done with a simple induction on the size of the proof which acts as hypothesis.
2. Similar to the previous item.
3. Similar to the previous item.
4. Assume $\mathcal{C}[e_1] \twoheadrightarrow t$, we can prove that then $\mathcal{C}[e_1\ ?\ e_2] \twoheadrightarrow t$ with a simple induction on the size of the proof for $\mathcal{C}[e_1] \twoheadrightarrow t$. Regarding the second part of this item, assume $\mathcal{C}[a_1\ ?\ \dots\ ?\ a_n] \twoheadrightarrow t$, then by the previuos items we may eliminate repeated elements in the chains and arrange them in way such that $b_1\dots b_m \equiv a_1\dots a_n b'_1\dots b'_k$ con $n+k=m$. Then by the first part of this item $[\![\mathcal{C}[a_1\ ?\ \dots\ ?\ a_n]]\!]^{pl} \subseteq [\![\mathcal{C}[a_1\ ?\ \dots\ ?\ a_n\ ?\ b'_1]]\!]^{pl} \subseteq \dots \subseteq [\![\mathcal{C}[a_1\ ?\ \dots\ ?\ a_n\ ?\ b'_1\ ?\ \dots\ ?\ b'_k]]\!]^{pl} = [\![\mathcal{C}[b_1\ ?\ \dots\ ?\ b_m]]\!]^{pl}$, and this process ends because both chains are finite.

Proof (For Theorem 1). First we will prove that for any $t \in CTerm_\perp$, if $\mathcal{C}[e] \twoheadrightarrow t$ then $\exists\{s_1,\dots,s_n\} \subseteq [\![e]\!]^{pl}$ such that $\mathcal{C}[s_1\ ?\ \dots\ ?\ s_n] \twoheadrightarrow t$, we proceed by induction on the size K of the proof for $\mathcal{C}[e] \twoheadrightarrow t$, measured as the number of rules of the calculus applied.

Base cases $K = 1$:

B Then we can take $\{s_1,\dots,s_n\} = \{\perp\}$ to do $\mathcal{C}[\perp] \twoheadrightarrow \perp$, by **B**.

RR, DC These cases correspond to $X \twoheadrightarrow X$ by **RR** and $c \twoheadrightarrow c$ by **DC**. Then $\mathcal{C} = []$ and so the hypothesis was $e \equiv \mathcal{C}[e] \twoheadrightarrow t$. Hence we can take $\{s_1,\dots,s_n\} = \{t\}$ to do $\mathcal{C}[s_1\ ?\ \dots\ ?\ s_n] \equiv [t] \equiv t \twoheadrightarrow t$, by Lemma 3.

Inductive steps $K > 1$:

DC If $\mathcal{C} = []$ then we are done like in the previous step. Otherwise we have:

$$\frac{e_1 \twoheadrightarrow t_1\ \dots\ \mathcal{C}'[e] \twoheadrightarrow t'\ \dots\ e_l \twoheadrightarrow t_l}{\mathcal{C}[e] \equiv c(e_1,\dots,\mathcal{C}'[e],\dots,e_l) \twoheadrightarrow c(t_1,\dots,t',\dots,t_l) \equiv t}\ DC$$

Then by IH $\exists\{s_1,\ldots,s_n\} \subseteq [\![e]\!]^{pl}$ such that $\mathcal{C}'[s_1 \ ? \ \ldots \ ? \ s_n] \twoheadrightarrow t'$, therefore we can build the following proof:

$$\frac{\dfrac{hypothesis}{e_1 \twoheadrightarrow t_1} \quad \ldots \quad \dfrac{IH}{\mathcal{C}'[s_1 \ ? \ \ldots \ ? \ s_n] \twoheadrightarrow t'} \quad \ldots \quad \dfrac{hypothesis}{e_l \twoheadrightarrow t_l}}{\mathcal{C}[s_1 \ ? \ \ldots \ ? \ s_n] \equiv c(e_1,\ldots,\mathcal{C}'[s_1 \ ? \ \ldots \ ? \ s_n],\ldots,e_l) \twoheadrightarrow c(t_1,\ldots,t',\ldots,t_l) \equiv t}\ DC$$

POR If $\mathcal{C} = []$ then we are done like in the previous step. Otherwise we have:

$$\frac{\begin{array}{ccccccc} e_1 \twoheadrightarrow p_1\theta_{11} & & \mathcal{C}'[e] \twoheadrightarrow p'\theta'_1 & & e_l \twoheadrightarrow p_l\theta_{l1} & & \\ \ldots & \ldots & \ldots & \ldots & \ldots & & \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & \mathcal{C}'[e] \twoheadrightarrow p'\theta'_{m'} & & e_l \twoheadrightarrow p_l\theta_{lm_l} & & r\theta \twoheadrightarrow t \end{array}}{\mathcal{C}[e] \equiv f(e_1,\ldots,\mathcal{C}'[e],\ldots,e_l) \twoheadrightarrow t}\ POR$$

with $\theta = ?(\theta_{11}\ldots\theta_{1m_1}) \uplus \ldots \uplus ?(\theta'_1\ldots\theta'_{m'}) \uplus \ldots \uplus ?(\theta_{l1}\ldots\theta_{lm_l})$ for some $(f(p_1,\ldots,p',\ldots,p_l) \to r) \in \mathcal{P}$. Then by IH for each $\theta'_i \in \{\theta'_1,\ldots,\theta'_{m'}\}$ $\exists\{s_{i1},\ldots,s_{in_i}\} \subseteq [\![e]\!]^{pl}$ such that $\mathcal{C}'[s_{i1} \ ? \ \ldots \ ? \ s_{in_i}] \twoheadrightarrow p'\theta'_i$, hence $\mathcal{C}'[s_{11} \ ? \ \ldots \ ? \ s_{1n_1} \ ? \ \ldots \ ? \ s_{m'1} \ ? \ \ldots \ ? \ s_{m'n_{m'}}] \twoheadrightarrow p'\theta'_i$ by Lemma 4. Therefore we can take:

$$\begin{array}{c} \{s_1,\ldots,s_n\} = \{s_{11} \ ? \ \ldots \ ? \ s_{1n_1} \ ? \ \ldots \ ? \ s_{m'1} \ ? \ \ldots \ ? \ s_{m'n_{m'}}\} \\ a \equiv \mathcal{C}'[s_{11} \ ? \ \ldots \ ? \ s_{1n_1} \ ? \ \ldots \ ? \ s_{m'1} \ ? \ \ldots \ ? \ s_{m'n_{m'}}] \end{array}$$

to build the following proof:

$$\frac{\begin{array}{ccccccc} e_1 \twoheadrightarrow p_1\theta_{11} & & a \twoheadrightarrow p'\theta'_1 & & e_l \twoheadrightarrow p_l\theta_{l1} & & \\ \ldots & \ldots & \ldots & \ldots & \ldots & & \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & a \twoheadrightarrow p'\theta'_{m'} & & e_l \twoheadrightarrow p_l\theta_{lm_l} & & r\theta \twoheadrightarrow t \end{array}}{\mathcal{C}[s_1 \ ? \ \ldots \ ? \ s_n] \equiv f(e_1,\ldots,a,\ldots,e_l) \twoheadrightarrow t}\ POR$$

Now we have to prove the other implication, that is, given $\{s_1,\ldots,s_n\} \subseteq [\![e]\!]^{pl}$ such that $\mathcal{C}[s_1 \ ? \ \ldots \ ? \ s_n] \twoheadrightarrow t$ then $\mathcal{C}[e] \twoheadrightarrow t$. If $\mathcal{C}[]$ then the hypothesis is $s_1 \ ? \ \ldots \ ? \ s_n \twoheadrightarrow t$, so it must exists some $s_i \in \{s_1,\ldots,s_n\}$ such that $s_i \twoheadrightarrow t$. But then $t \sqsubseteq s_i$ by Lemma 3, as $s_i \in CTerm_\perp$, as it is a value for e. But then the hypothesis $e \twoheadrightarrow s_i$ and $t \sqsubseteq s_i$ implies $e \twoheadrightarrow t$ by the monotonicity of Lemma 1.
To prove the case when $\mathcal{C} \neq []$ we need to do a simple induction on the size of $\mathcal{C}[s_1 \ ? \ \ldots \ ? \ s_n] \twoheadrightarrow t$, in which we will not assume $\mathcal{C} \neq []$.

Some facts about the preorders $\sqsubseteq_\pi$ and $\trianglelefteq$:

Lemma 5.

1. *$\forall\theta,\theta' \in CSusbt_\perp$, $\theta \sqsubseteq \theta'$ iff $\theta \sqsubseteq_\pi \theta'$.*
2. *$\sqsubseteq_\pi$: $CSubst^?_\perp \times CSubst^?_\perp$ is a preorder but not a partial order.*
3. *Given $\theta,\theta' \in CSubst^?_\perp$ if $\theta \sqsubseteq_\pi \theta'$ then $\theta \trianglelefteq \theta'$*
4. *$\trianglelefteq : Subst_\perp \times Subst_\perp$ is a preorder but not a partial order*

Proof.

1. By definition.
2. It is very easy to check that it is reflexive and transitive, but it is not antisymetric, as $[X/0] \sqsubseteq_\pi [X/0 \ ? \ 0]$, $[X/0 \ ? \ 0] \sqsubseteq_\pi [X/0]$ but $[X/0] \neq [X/0 \ ? \ 0]$.
3. For any $X \in \mathcal{V}$, if $\theta(X) = t_1 \ ? \ \ldots \ ? \ t_n$ and $\theta'(X) = t'_1 \ ? \ \ldots \ ? \ t'_m$ (note that $\theta(X)$ has that shape even when $X \notin dom(\theta)$, as X has that shape) and $\theta(X) \twoheadrightarrow t$ then it must exist some $t_i \in \{t_1,\ldots,t_n\}$ such that $t_i \twoheadrightarrow t$, and so $t \sqsubseteq t_i$ by Lemma 3. But then $t \sqsubseteq t_i \sqsubseteq t'_j$ for some $t_j \in \{t'_1,\ldots,t'_m\}$, because $\theta \sqsubseteq_\pi \theta'$, hence $t'_j \twoheadrightarrow t$ by Lemma 3 and so $\theta'(X) \twoheadrightarrow t$.
4. It is very easy to check that it is reflexive and transitive, but it is not antisymetric, because given $\mathcal{P} = \{f \to 1, g \to 1\}$ we have $[X/f] \trianglelefteq [X/g]$ and $[X/g] \trianglelefteq [X/f]$ while $[X/f] \neq [X/g]$.

Proof (For Lemma 2).

1. If $e \equiv X \in \mathcal{V}$, assume $e\sigma \equiv \sigma(X) \twoheadrightarrow t$, then $e\sigma' \equiv \sigma'(X) \twoheadrightarrow t$ with a proof of the same size or smaller, by hypothesis. Otherwise we proceed by induction on the structure of $e\sigma \twoheadrightarrow t$.
   Base cases
   B Then $t \equiv \bot$ and $e\sigma' \twoheadrightarrow \bot$ with a proof of size 1 just applying rule B.
   RR Then $e \in \mathcal{V}$ and we are in the previous case.
   DC Then $e \equiv c \in CS^0$, as $e \notin \mathcal{V}$, hence $e\sigma \equiv c \equiv e\sigma'$ and every proof for $e\sigma \twoheadrightarrow t$ is a proof for $e\sigma' \twoheadrightarrow t$.
   Inductive steps
   DC Then $e \equiv c(e_1, \ldots, e_n)$, as $e \notin \mathcal{V}$, and we have:

$$\frac{e_1\sigma \twoheadrightarrow t_1 \ \ldots \ e_n\sigma \twoheadrightarrow t_n}{e\sigma \equiv c(e_1\sigma, \ldots, e_n\sigma) \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t} \ DC$$

   By IH or the proof of the other cases $\forall i \in \{1, \ldots, n\}$ we have $e_i\sigma' \twoheadrightarrow t_i$ with a proof of the same size or smaller, so we can built a proof for $e\sigma' \equiv c(e_1\sigma', \ldots, e_n\sigma') \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t$ using DC, with a size equal or smaller than the size of the starting proof.
   OR Very similar to the proof of the previuos case. We also have $e \equiv f(e_1, \ldots, e_n)$ (as $e \notin \mathcal{V}$) and given a proof for $e\sigma \equiv f(e_1, \ldots, e_n)\sigma \twoheadrightarrow t$, we apply the IH to every $e_i\sigma \twoheadrightarrow p_i\theta_{ij}$ to get that $e_i\sigma' \twoheadrightarrow p_i\theta_{ij}$ with a proof of the same size or smaller. But then we can use this proofs in a POR step from $e\sigma' \equiv f(e_1\sigma', \ldots, e_n\sigma')$ and use the same substitution $\theta \in CSubst^?_\bot$ for parameter passing, constructing a proof with a size equal or smaller than the size of the starting one.
2. If $\theta \sqsubseteq \theta'$ then for any $X \in \mathcal{V}$ we have $\theta(X) \sqsubseteq \theta'(X)$, hence if $\theta(X) \twoheadrightarrow t$ then $\theta'(X) \twoheadrightarrow t$ by the mononicity of $\pi CRWL$. But then we can apply the strong monotonicity of $Subst_\bot$ to get the desired result.
3. Using the notations of Definition 3, given $X_i \in \overline{X} = var(e)$ if the set of positions of the occurrences of X_i in e is $\{p_{i1}, \ldots, p_{im_i}\}$ then $e \equiv e[X_i]_{p_{i1}} \equiv (e[X_i]_{p_{i1}})[X_i]_{p_{i2}} \equiv \ldots e[X_i]_{p_{i1}} \ldots [X_i]_{p_{im_i}}$. As the positions of any pair of different occurrences of (possibly different) variables are parallel, we can do this for every variable in $\overline{X}$ to get $e \equiv e[Y_1]_{o_1} \ldots [Y_m]_{o_m}$, where $\{o_1, \ldots, o_m\}$ is the set of positions of every occurrence in e of any variable in $var(e)$ and $\{Y_1, \ldots, Y_m\} = \overline{X}$. Note how each position in $\{o_1, \ldots, o_m\}$ is parallel to each other. But then we can apply the recasted version of Theorem 1 that appears in Definition 3, to get:

$$\begin{array}{lll}
[\![e\sigma]\!]^{pl} = [\![e[Y_1\sigma]_{o_1} \ldots [Y_m\sigma]_{o_m}]\!]^{pl} & & \\
= \bigcup_{\overline{t_1} \subseteq [\![Y_1\sigma]\!]^{pl}} [\![e[?\overline{t_1}]_{o_1} \ldots [Y_m\sigma]_{o_m}]\!]^{pl} & & \text{by Theorem 1} \\
= \bigcup_{\overline{t} \subseteq [\![Y\sigma]\!]^{pl}} [\![e[?\overline{\overline{t}}]_{\overline{o}}]\!]^{pl} & & \text{by Theorem 1 (many times)} \\
= \bigcup_{\overline{t} \subseteq [\![Y\sigma']\!]^{pl}} [\![e[?\overline{\overline{t}}]_{\overline{o}}]\!]^{pl} & & \text{as } \sigma \trianglelefteq \sigma' \\
= [\![e[Y_1\sigma']_{o_1} \ldots [Y_m\sigma']_{o_m}]\!]^{pl} = [\![e\sigma']\!]^{pl} & & \text{by Theorem 1}
\end{array}$$

   Note that we cannot claim $e\sigma' \twoheadrightarrow t$ with proof of the same size of smaller, as we can see for example with $\sigma = [X/0] \trianglelefteq [X/0 \ ? \ 0] = \sigma', e \equiv X, t \equiv 0$ for which $e\sigma \equiv 0 \twoheadrightarrow 0$ with size one but $e\sigma' \equiv 0 \ ? \ 0 \twoheadrightarrow 0$ with size greater or equal to four.
4. If $\theta \sqsubseteq_\pi \theta'$ then $\theta \trianglelefteq \theta'$ by Lemma 5, hence this item holds by the previous item.

A.2 For Section 4

Lemma 6. *For any CRWL-program $\mathcal{P}$, $e \in Exp_\bot, t \in CTerm_\bot, \theta \in CSubst_\bot$ if $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ then $\mathcal{P} \vdash_{\pi CRWL} e\theta \twoheadrightarrow t\theta$.*

Proof. If $t \equiv \bot$ then $e\theta \twoheadrightarrow t\theta \equiv \bot$ by B, otherwise the proof is just a simple induction over the size of $e \twoheadrightarrow t$.

In order to prove the soundness of a rewriting step wrt. $\pi CRWL$ we will need the following auxiliary but revealing results.

Definition 4 (Denotation of substitutions). *For any CRWL-program $\mathcal{P}$, $\sigma \in Subst_\bot$ we define* $[\![\sigma]\!]^{pl}_{\mathcal{P}} = \{\theta \in CSubst_\bot \mid \forall X \in \mathcal{V},\ \mathcal{P} \vdash_{\pi CRWL} \sigma(X) \twoheadrightarrow \theta(X)\}$.

The denotations of substitutions enjoys the following interesting properties:

Lemma 7. *For any CRWL-program $\mathcal{P}$, $\sigma \in Subst_\bot$*

a) $[\![\sigma]\!]^{pl} \neq \emptyset$ *and given* $\overline{X} = dom(\sigma)$ *then* $[\overline{X/\ \bot}] \in [\![\sigma]\!]^{pl}$.
b) $[\![\sigma]\!]^{pl}$ *is an infinite set.*
c) Given $\theta \in [\![\sigma]\!]^{pl}$ *this does not imply neither* $dom(\theta) \subseteq dom(\sigma)$ *nor* $dom(\sigma) \subseteq dom(\theta)$

Proof.

a) It is enough to prove that if $\overline{X} = dom(\sigma)$ then $[\overline{X/\ \bot}] \in [\![\sigma]\!]^{pl}$. First of all $[\overline{X/\ \bot}] \in CSubst_\bot$ by definition. Now consider some $Y \in \mathcal{V}$.
   i) If $Y \in \overline{X}$ then $\vdash_{\pi CRWL} \sigma(Y) \twoheadrightarrow \bot \equiv Y[\overline{X/\ \bot}]$, by rule B.
   ii) Otherwise $Y \notin \overline{X} = dom(\sigma)$, hence $\vdash_{\pi CRWL} \sigma(Y) \equiv Y \twoheadrightarrow Y \equiv Y[\overline{X/\ \bot}]$, by rule RR.
b) By *a)* we know there exists at least one $\theta \in [\![\sigma]\!]^{pl}$. Besides, as substitutions are finite mappings we can take some $X \in \mathcal{V}$ such that $X \notin dom(\theta)$. Hence $\theta \uplus [X/\ \bot]$ is correctly defined, we will also see that $(\theta \uplus [X/\ \bot]) \in [\![\sigma]\!]^{pl}$. First of all $\theta \in [\![\sigma]\!]^{pl}$ implies $\theta \in CSubst_\bot$, so $\theta \uplus [X/\ \bot] \in CSubst_\bot$ too. Besides for any $Y \in \mathcal{V}$:
   i) If $Y \equiv X$ then $\vdash_{\pi CRWL} \sigma(Y) \twoheadrightarrow \bot \equiv (\theta \uplus [X/\ \bot])(X) \equiv (\theta \uplus [X/\ \bot])(Y)$, by rule B.
   ii) Otherwise $Y \not\equiv X$ and then $\theta \in [\![\sigma]\!]^{pl}$ implies $\vdash_{\pi CRWL} \sigma(Y) \twoheadrightarrow \theta(Y) \equiv (\theta \uplus [X/\ \bot])(Y)$, as $Y \not\equiv X$.
c) Taking $\sigma = \epsilon$ and $\theta = [X/\ \bot]$ we have $\theta \in [\![\sigma]\!]^{pl}$ while $dom(\theta) \not\subseteq dom(\sigma)$. On the other hand under the program $\{f(X) \to X\}$ we can take $\sigma = [X/f(X)]$ and $\theta = \epsilon$ for which $\theta \in [\![\sigma]\!]^{pl}$ while $dom(\sigma) \not\subseteq dom(\theta)$.

Note that part *b)* from Lemma 7 is not so strange as it might look at a first sight, because it is easy to see that $\forall \sigma \in Subst_\bot$ we have that $\{\sigma' \in Subst_\bot \mid \sigma' \sqsubseteq \sigma\}$ is an infinite set too.

Lemma 8. *For any* $\sigma \in Subst_\bot, e \in Exp_\bot, t \in CTerm_\bot$ *if* $\vdash_{\pi CRWL} e\sigma \twoheadrightarrow t$ *then* $\exists \Theta \subseteq [\![\sigma]\!]^{pl}$ *finite and not empty such that* $\vdash_{\pi CRWL} e(?\Theta) \twoheadrightarrow t$

Proof. By a case distintion over e:

– If $e \equiv X \in dom(\sigma)$: Then $e\sigma \equiv \sigma(X) \twoheadrightarrow t$, so we can define:

$$\theta(Y) = \begin{cases} t & \text{if } Y \equiv X \\ \bot & \text{if } Y \in (dom(\sigma) \setminus \{X\}) \\ Y & \text{if } Y \notin dom(\sigma) \end{cases}$$

Then $\theta \in [\![\sigma]\!]^{pl}$ because obviously $\theta \in CSusbt_\bot$, and given $Z \in \mathcal{V}$.
   a) If $Z \equiv X$ then $\vdash_{\pi CRWL} \sigma(Z) \equiv \sigma(X) \twoheadrightarrow t \equiv \theta(Z)$ by hypothesis.
   b) If $Z \in (dom(\sigma) \setminus \{X\})$ then $\vdash_{\pi CRWL} \sigma(Z) \twoheadrightarrow \bot \equiv \theta(Z)$ by rule B.
   c) Otherwise $Z \notin dom(\sigma)$ and then $\vdash_{\pi CRWL} \sigma(Z) \equiv Z \twoheadrightarrow Z \equiv \theta(Z)$ by rule RR.
Now we can take $\Theta = \{\theta\}$ for which $e(?\Theta) \equiv \theta(X) \equiv t \twoheadrightarrow t$, by Lemma 3.
– If $e \equiv X \notin dom(\sigma)$: Then given $\overline{Y} = dom(\sigma)$ we define $[\overline{Y/\ \bot}]$ for which $[\overline{Y/\ \bot}] \in [\![\sigma]\!]^{pl}$ by part *a)* of Lemma 7, so we can take $\Theta = \{[\overline{Y/\ \bot}]\}$ for which $[\![e\sigma]\!]^{pl} = [\![X]\!]^{pl} = [\![X(?\Theta)]\!]^{pl}$.
– If $e \notin \mathcal{V}$ then we proceed by induction over the structure of $e\sigma \twoheadrightarrow t$:
Base cases
   B Then $t \equiv \bot$, so given $\overline{Y} = dom(\sigma)$ we can take $\Theta = \{[\overline{Y/\ \bot}]\}$ for which $e(?\Theta) \twoheadrightarrow \bot$ by B.
   RR Then $e \in \mathcal{V}$ and we are in the previous case.
   DC Similar to the case for $e \equiv X \notin dom(\sigma)$.
Inductive steps

DC Then $e \equiv c(e_1, \ldots, e_n)$, as $e \notin \mathcal{V}$, and we have:

$$\frac{e_1\sigma \twoheadrightarrow t_1 \ \ldots \ e_n\sigma \twoheadrightarrow t_n}{e\sigma \equiv c(e_1\sigma, \ldots, e_n\sigma) \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t}\ DC$$

By IH or the proof of the other cases $\forall i \in \{1, \ldots, n\}\ \exists \Theta_i \subseteq [\![\sigma]\!]^{pl}$ such that $e_i(?\Theta_i) \twoheadrightarrow t_i$. Hence we can define $\Theta = \bigcup_{i\in\{1,\ldots,n\}} \Theta_i$, for which is trivial to prove that $\forall i \in \{1, \ldots, n\}\ ?\Theta_i \sqsubseteq_\pi ?\Theta$, and so $\forall i \in \{1, \ldots, n\}\ e_i(?\Theta_i) \twoheadrightarrow t_i$ implies $e_i(?\Theta) \twoheadrightarrow t_i$, by Lemma 2. Therefore $e(?\Theta) \twoheadrightarrow c(t_1, \ldots, t_n)$ by DC.

POR Very similar to the proof of the previuos case. We also have $e \equiv f(e_1, \ldots, e_n)$ (as $e \notin \mathcal{V}$) and given a proof for $e\sigma \equiv f(e_1, \ldots, e_n)\sigma \twoheadrightarrow t$, we apply the IH or the proof of the other cases to every $e_i\sigma \twoheadrightarrow p_i\theta_{ij}$ to get some $\Theta_{ij} \subseteq [\![\sigma]\!]^{pl}$ such that $e_i(?\Theta_{ij}) \twoheadrightarrow p_i\theta_{ij}$. Then we define $\Theta = \bigcup_{i\in\{1,\ldots,n\}} \bigcup_{j\in\{1,\ldots,m_i\}} \Theta_{ij}$ for which $\forall i, j,\ ?\Theta_{ij} \sqsubseteq_\pi ?\Theta$ obviously holds, and as a consequence $\forall i, j,\ e_i(?\Theta) \twoheadrightarrow p_i\theta_{ij}$. Hence with $e(?\Theta) \equiv f(e_1(?\Theta), \ldots, e_n(?\Theta))$ we can compute the same value for its arguments and thus use the same substitution $\theta \in CSubst^?_\perp$ for parameter passing in POR.

Lemma 9. *For any finite not empty $\Theta \subseteq [\![\sigma]\!]^{pl}$ we have $?\Theta \trianglelefteq \sigma$.*

Proof. First of all Θ is required to be finite and not empty because otherwise $?\Theta$ is not defined. Given $X \in \mathcal{V}$:

a) If $X \in dom(?\Theta)$: assume $(?\Theta)(X) \twoheadrightarrow t$ then as $X \in dom(?\Theta)$ we have $\exists \theta_i \in \Theta$ such that $X \in dom(\theta_i)$ and $\theta_i(X) \twoheadrightarrow t$, hence $t \sqsubseteq \theta_i(X)$ by Lemma 3. But as $\Theta \subseteq [\![\sigma]\!]^{pl}$ then $\theta_i \in [\![\sigma]\!]^{pl}$, therefore $\sigma(X) \twoheadrightarrow \theta_i(X)$, and so $\sigma(X) \twoheadrightarrow t$ by the monotonicity Lemma 1, as $t \sqsubseteq \theta_i(X)$.
b) Otherwise $X \notin dom(?\Theta)$, then assume $(?\Theta)(X) \equiv X \twoheadrightarrow t$, which implies either $t \equiv \perp$ or $t \equiv X$. The case for $t \equiv \perp$ is trivial by using rule OR. On the other hand if $t \equiv X$ as $X \notin dom(?\Theta)$ then $\forall \theta_i \in \Theta$ we have $X \notin dom(\theta_i)$. We can take some $\theta_i \in \Theta$—there must be someone as Θ is not empty by hypothesis—and then as $\Theta \subseteq [\![\sigma]\!]^{pl}$ we have $\theta_i \in [\![\sigma]\!]^{pl}$ and so $\vdash_{\pi CRWL} \sigma(X) \twoheadrightarrow \theta_i(X) \equiv X \equiv t$.

Lemma 10. *For any $t \in CTerm_\perp, \Theta \subseteq CSusbt_\perp$ finite, given $\theta_i \in \Theta$ then $\vdash_{\pi CRWL} t(?\Theta) \twoheadrightarrow t\theta_i$*

Proof. A simple induction on the structure of t.

Lemma 11 (One step soundness of $\rightarrow$ wrt $\pi CRWL$). *For any CRWL-program $\mathcal{P}$, $e, e' \in Exp$ if $e \rightarrow e'$ then $[\![e']\!]^{pl} \subseteq [\![e]\!]^{pl}$.*

Proof (For Lemma 11). Assume the step has been performed at the top of the expression, that is, $e \equiv f(p_1, \ldots, p_n)\sigma \rightarrow r\sigma \equiv e'$ for some $(f(p_1, \ldots, p_n) \rightarrow r) \in \mathcal{P}$. Given some $t \in CTerm_\perp$ such that $r\sigma \twoheadrightarrow t$ then by Lemma 8 there must exists some $\Theta \subseteq [\![\sigma]\!]^{pl}$ not empty and finite for which $r(?\Theta) \twoheadrightarrow t$. If $\Theta = \{\theta_1, \ldots, \theta_m\}$ then we can do:

$$\dfrac{\dfrac{\dfrac{}{p_1(?\Theta) \twoheadrightarrow p_1\theta_1 \equiv p_1\theta_1|_{var(p_1)}}\ \text{Lemma 10}}{\ldots}\ \ldots\ \dfrac{\dfrac{}{p_n(?\Theta) \twoheadrightarrow p_n\theta_1 \equiv p_n\theta_1|_{var(p_n)}}\ \text{Lemma 10}}{\ldots} \qquad r\theta' \equiv_{(*)} r(?\Theta) \twoheadrightarrow t}{f(p_1, \ldots, p_n)(?\Theta) \twoheadrightarrow t}\ POR$$

with $\theta' = ?\{\theta_1|_{var(p_1)} \ldots \theta_m|_{var(p_1)}\} \uplus \ldots \uplus\ ?\{\theta_1|_{var(p_n)} \ldots \theta_m|_{var(p_n)}\}$, using the same program rule. The equivalence $(*)$ holds because for any $X \in var(r) \subseteq var(f(p_1, \ldots, p_n))$ there must exist exactly one p_i such that $X \in var(p_i)$, because of left linearity. Hence

$$\begin{aligned} X\theta' &\equiv X(?\{\theta_1|_{var(p_i)} \ldots \theta_m|_{var(p_i)}\}) \\ &\equiv \theta_1(X)\ ?\ \ldots\ ?\ \theta_m(X) \equiv (?\Theta)(X) \end{aligned}$$

Now we can apply Lemma 9 to get $?\Theta \trianglelefteq \sigma$, which combined with Lemma 2 implies $f(p_1, \ldots, p_n)\sigma \twoheadrightarrow t$.
If the step was not performed at the root of the expression then we have $e \equiv \mathcal{C}[f(\overline{p})\sigma] \rightarrow \mathcal{C}[r\sigma] \equiv e'$ for which

$f(\overline{p})\sigma \to r\sigma$ is performed at the top. But then $[\![r\sigma]\!]^{pl} \subseteq [\![f(\overline{p})\sigma]\!]^{pl}$ by the proof of the previous case, and we can chain:

$$[\![\mathcal{C}[r\sigma]]\!]^{pl} = \bigcup_{\{t_1,\ldots,t_n\}\subseteq[\![r\sigma]\!]^{pl}} \quad \text{by Theorem 1}$$
$$\subseteq \bigcup_{\{t_1,\ldots,t_n\}\subseteq[\![f(\overline{p})\sigma]\!]^{pl}} = [\![\mathcal{C}[f(\overline{p})\sigma]]\!]^{pl} \quad \text{by Theorem 1}$$

Now we have the tools to prove Theorem 4 and Corollary 1.

Proof (For Theorem 4). Given some $t \in [\![e]\!]^{rw}$, by definition $\exists e' \in Exp$ such that $t \sqsubseteq |e'|$ and $e \to^* e'$. We can extend Lemma 11 to $\to^*$ by a simple induction on the length of $e \to^* e'$, hence $[\![e']\!]^{pl} \subseteq [\![e]\!]^{pl}$. As $\forall e \in Exp_\bot, |e| \in [\![e]\!]^{pl}$ (by a simple induction on the structure of e), then $t \sqsubseteq |e'| \in [\![e']\!]^{pl} \subseteq [\![e]\!]^{pl}$, hence $t \in [\![e]\!]^{pl}$ by Lemma 1. Example 5 shows that the converse inclusion does not hold in general.

Proof (For Corollary 1). The first part holds just combining Theorem 3 with Theorem 4. Concerning the second part, assume $\vdash_{CRWL} e \twoheadrightarrow t$, in other words, $t \in [\![e]\!]^{sg}$. Then by the first part $t \in [\![e]\!]^{rw}$, hence $e \to^* e'$ such that $t \sqsubseteq |e'|$. But as $t \in CTerm$ then t is maximal wrt $\sqsubseteq$ (a know property of $\sqsubseteq$), and so $t \sqsubseteq |e'|$ implies $t \equiv |e'|$, which implies $t \equiv e'$, as t is total (very easy to check by induction on the structure of t). Therefore $e \to^* e' \equiv t$. Concerning the last fact holds by defition, if $e \to^* t \in CTerm$ then $t \in [\![e]\!]^{rw}$ by definition, as $t \sqsubseteq t \equiv |t|$ (an old property of shells easy to check by induction on the structure of t), but then $t \in [\![e]\!]^{pl}$ by the first part, in other words, $e \twoheadrightarrow t$.

A.3 For Section 5

The following auxiliary results will be needed to prove Theorem 5.

Lemma 12. *For any CRWL-program $\mathcal{P}$, πCRWL-statement $if\ e_1\ then\ e_2 \twoheadrightarrow t$ there is a πCRWL-proof for that statement of the shape:*

$$\frac{e_1 \twoheadrightarrow true \quad e_2 \twoheadrightarrow t \quad t \twoheadrightarrow t}{if\ e_1\ then\ e_2 \twoheadrightarrow t}\ POR$$

Proof. As the only rule for $if\_then\_$ is $if\ true\ then\ X \to X$ then every proof must be of the shape:

$$\frac{\begin{array}{lll} e_1 \twoheadrightarrow true\theta_1 & e_2 \twoheadrightarrow t_1 & \\ \ldots & \ldots & t_1?\ldots?t_l \twoheadrightarrow t \\ e_1 \twoheadrightarrow true\theta_m & e_2 \twoheadrightarrow t_l & \end{array}}{if\ e_1\ then\ e_2 \twoheadrightarrow t}\ POR$$

As $t_1?\ldots?t_l \twoheadrightarrow t$ there must exist some $t_i \in \{t_1,\ldots,t_l\}$ such that $t_i \twoheadrightarrow t$, but then $t \sqsubseteq t_i$ by Lemma 3 and so $e_2 \twoheadrightarrow t_i$ implies $e_2 \twoheadrightarrow t$ by Lemma 1, and we can apply Lemma 3 again to get $t \twoheadrightarrow t$ and construct the proof of the formulation.

Proof (For Theorem 5). Assume $pST(\mathcal{P}) \vdash_{\pi CRWL} e \twoheadrightarrow t$ for some $t \in CTerm_\bot$, we will see that then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ by induction on the size of $pST(\mathcal{P}) \vdash_{\pi CRWL} e \twoheadrightarrow t$. The base cases are trivial because no program rule is involved, and so it is the case for DC, in which we only have to apply the IH over the hypothesis. The case for POR when $e \equiv f(\overline{e})$ and $f \in \{\_?\_, if\_then\_\}$ can be resolved applying the IH too, so the difficult case is that in which $f \in \{\_?\_, if\_then\_\}$. For the sake of sake of simplicity we will consider $f \in FS^1$, the proof can be easily extended to functions with zero or more than one arguments. Assume the rule used was $f(Y) \to if\ match(Y)\ then\ r[\overline{X_j/project_j(Y)}]$, corresponding to the original rule $f(p) \to r$ and with the auxiliary functions defined by $match(p) \to true$, $project_j(p) \to X_j$, for $\overline{X_j} = var(p) \cap var(r)$. Then the proof was of the shape:

$$\frac{\begin{array}{l} e \twoheadrightarrow t_1 \\ \ldots \\ e \twoheadrightarrow t_m \quad if\ match(t_1?\ldots?t_m)\ then\ r[\overline{X_j/project_j(t_1?\ldots?t_m)}] \twoheadrightarrow t \end{array}}{pST(\mathcal{P}) \vdash_{\pi CRWL} f(e) \twoheadrightarrow t}\ POR$$

where

$$\dfrac{\dfrac{t_1?\ldots?t_m \twoheadrightarrow p\mu \quad true \twoheadrightarrow true}{match(t_1?\ldots?t_m) \twoheadrightarrow true}\ POR \quad r[\overline{X_j/project_j(t_1?\ldots?t_m)}] \twoheadrightarrow t \quad t \twoheadrightarrow t}{pST(\mathcal{P}) \vdash_{\pi CRWL} if\ match(t_1?\ldots?t_m)\ then\ r[\overline{X_j/project_j(t_1?\ldots?t_m)}] \twoheadrightarrow t}\ POR$$

by Lemma 12. Let $s_1 \ldots s_l = t_1 \ldots t_m \mid \lambda t.(t \equiv p\theta)$ with $dom(\theta) = var(p)$, for some $\theta \in CSubst_\perp$, one θ for each s in $s_1 \ldots s_n$. Then $pST(\mathcal{P}) \vdash_{\pi CRWL} t_1?\ldots?t_m \twoheadrightarrow p\mu$ implies $\exists j \in \{1,\ldots,m\}$ such that $pST(\mathcal{P}) \vdash_{\pi CRWL} t_j \twoheadrightarrow p\mu$, and so $p\mu \sqsubseteq t_j$ by Lemma 3. But then it is very easy to prove by induction on the structure of t, taking advantage of its linearity and totality, that there must exist some $\theta_j \in CSubst_\perp$ such that $t_j \equiv p\theta_j$. Hence $s_1 \ldots s_l$ is not empty and then it is very easy to prove that $[\overline{X_j/project_j(t_1?\ldots?t_m)}]$ and $[\overline{X_j/project_j(s_1?\ldots?s_l)}]$ verify the conditions to apply the strong monotonicity of Lemma 2 in order to get that $pST(\mathcal{P}) \vdash_{\pi CRWL} r[\overline{X_j/project_j(s_1?\ldots?s_l)}] \twoheadrightarrow t$ with a proof of the same size or smaller. By definition $s_1 \ldots s_l \equiv p\theta_1 \ldots p\theta_l$, now we will see that the substitutions $[\overline{X_j/project_j(s_1?\ldots?s_l)}]$ and $?\{\theta_1,\ldots,\theta_l\}|_{var(r)}$ also verify the conditions of to apply the strong monotonicity of Lemma 2. As $\forall\theta \in \{\theta_1,\ldots,\theta_l\}\ dom(\theta) = var(p)$ then $dom(?\{\theta_1,\ldots,\theta_l\}|_{var(r)}) = \overline{X_j}$, so given $X \notin \overline{X_j}$ both substitutions leave it untouched. On the other hand if $X \equiv X_j \in \overline{X_j}$, given

$$\dfrac{\begin{array}{l} s_1?\ldots?s_l \twoheadrightarrow p\mu_1 \\ \ldots \\ s_1?\ldots?s_l \twoheadrightarrow p\mu_h \quad X_j(?\{\mu_1,\ldots,\mu_h\}) \twoheadrightarrow t \end{array}}{pST(\mathcal{P}) \vdash_{\pi CRWL} X_j[\overline{X_j/project_j(s_1?\ldots?s_l)}] \equiv project_j(s_1?\ldots?s_l) \twoheadrightarrow t}\ POR$$

Then $pST(\mathcal{P}) \vdash_{\pi CRWL} X_j(?\{\mu_1,\ldots,\mu_h\}) \twoheadrightarrow t$ implies $pST(\mathcal{P}) \vdash_{\pi CRWL} \mu(X_j) \twoheadrightarrow t$ for some $\mu \in \{\mu_1,\ldots,\mu_h\}$. But then $pST(\mathcal{P}) \vdash_{\pi CRWL} s_1?\ldots?s_l \twoheadrightarrow p\mu$, and so $pST(\mathcal{P}) \vdash_{\pi CRWL} s \twoheadrightarrow p\mu$ for some $s \in \{s_1,\ldots,s_l\}$. Hence, by Lemma 3 $t \sqsubseteq \mu(X_j)$ and $p\mu \sqsubseteq s$, and as $s_1 \ldots s_l \equiv p\theta_1 \ldots p\theta_l$ by definition then $p\mu \sqsubseteq s \equiv p\theta$ for some $\theta \in \{\theta_1,\ldots,\theta_l\}$. But as $X_j \in \overline{X_j} \subseteq var(p)$ then $p\mu \sqsubseteq p\theta$ implies $\mu(X_j) \sqsubseteq \theta(X_j)$, hence $t \sqsubseteq \mu(X_j) \sqsubseteq \theta(X_j)$ and $pST(\mathcal{P}) \vdash_{\pi CRWL} \theta(X_j) \twoheadrightarrow t$ with a proof of the same size or smaller, by Lemma 1, and so $pST(\mathcal{P}) \vdash_{\pi CRWL} X_j(?\{\theta_1,\ldots,\theta_l\}|_{var(r)}) \twoheadrightarrow t$ with a proof of the same size or smaller than the proof for $pST(\mathcal{P}) \vdash_{\pi CRWL} X_j[\overline{X_j/project_j(s_1?\ldots?s_l)}] \twoheadrightarrow t$. But then we can apply Lemma 2 to get $pST(\mathcal{P}) \vdash_{\pi CRWL} r(?\{\theta_1,\ldots,\theta_l\}|_{var(r)}) \twoheadrightarrow t$ to which we can apply the IH to get $\mathcal{P} \vdash_{\pi CRWL} r(?\{\theta_1,\ldots,\theta_l\}|_{var(r)}) \twoheadrightarrow t$. We can also apply the IH to each $pST(\mathcal{P}) \vdash_{\pi CRWL} e_i \twoheadrightarrow t_l \equiv s_l \equiv p\theta$ and build the following proof:

$$\dfrac{\begin{array}{l} e \twoheadrightarrow s_1 \equiv p\theta_1 \\ \ldots \\ e \twoheadrightarrow s_l \equiv p\theta_l \quad r(?\{\theta_1,\ldots,\theta_l\}) \equiv r(?\{\theta_1,\ldots,\theta_l\}|_{var(r)}) \twoheadrightarrow t \end{array}}{\mathcal{P} \vdash_{\pi CRWL} f(e) \twoheadrightarrow t}\ POR$$

Concerning the proof for Theorem 6, we will use the following auxiliary results.

Lemma 13. *For every $e \in Exp$ and $p \in CTerm$ linear, given $\theta \in CSubst_\perp$ such that $dom(\theta) \subseteq FV(p)$, if $p\theta \sqsubseteq |e|$ then $\exists\sigma \in Subst$ such that $dom(\sigma) = dom(\theta)$, $p\sigma \equiv e$ and $\theta \sqsubseteq \sigma$.*

Proof. See [16].

Definition 5. *Given a signature $\Sigma = FS \uplus CS$ and a CRWL-program $\mathcal{P}$ the set $FS^{\mathcal{P}} \subseteq FS$ is defined as $FS^{\mathcal{P}} = \{f \in FS \mid \exists(f(\overline{p}) \to r) \in \mathcal{P}\}$*

Definition 6. *Given a πCRWL-proof Δ for a πCRWL-statement $e \twoheadrightarrow t$ by Π^Δ we denote the multiset of πCRWL-statements that compose Δ, including $e \twoheadrightarrow t$. Sometimes we will use $\Pi^{e \twoheadrightarrow t}$ when Δ is implicit. We will also use Δ_π to refer to the subproof for some premise π of Δ, when it is implicit.*

Lemma 14. *For any πCRWL-statement $e \twoheadrightarrow t$ which holds exists some πCRWL-proof Δ such that $\forall e' \twoheadrightarrow t' \in \Pi^\Delta$, $e' \twoheadrightarrow t'$ is not a premise, neither directly nor indirectly, of itself in Δ.*

Proof. As $e \twoheadrightarrow t$ holds we may assume some $\pi CRWL$-proof Δ for it. If no $e' \twoheadrightarrow t' \in \Pi^{\Delta}$ is premise of itself then we are done. Otherwise as any $\pi CRWL$-proof is finite then taking the subproof corresponding to some $e' \twoheadrightarrow t'$ which is premise of itself there must be some $e' \twoheadrightarrow t'$ which does not have $e' \twoheadrightarrow t'$ as its premise in its corresponding proof. Then we can use that proof to replace the subproof for $e' \twoheadrightarrow t'$, as the proof is finite this process ends, because each time the number of sentences premises of itself decreases.

Lemma 15. *Given a CRWL-program $\mathcal{P}$ let $\hat{\mathcal{P}} \uplus \mathcal{M} = pST(\mathcal{P})$, where $\mathcal{M}$ is the set containing the rules for $\_?\_, if\_then\_$ and the new functions match and project, and $\hat{\mathcal{P}}$ contains the new versions of the original rules of $\mathcal{P}$. Then for any $e \in Exp_{\perp}, t \in CTerm_{\perp}$ constructed using just symbols in the signature of $\mathcal{P} \uplus \mathcal{M}$ we have $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \twoheadrightarrow t$ implies $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \rightarrow^{*} e'$ such that $t \sqsubseteq |e'|$.*

Proof. For any proof for $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \twoheadrightarrow t$ we define $sa\mathcal{P}(e \twoheadrightarrow t)$ by

$$sa\mathcal{P}(e \twoheadrightarrow t) = \{e' \twoheadrightarrow t' \mid (e' \twoheadrightarrow t') \in \Pi^{e \twoheadrightarrow t} \wedge e' \equiv f(\overline{a}) \text{ for some } f \in FS^{\mathcal{P}}\}$$

Note that $sa\mathcal{P}(e \twoheadrightarrow t)$ is a set, not a multiset. Besides for any $\pi \in \Pi^{e \twoheadrightarrow t}$ we have $sa\mathcal{P}(\pi) \subseteq sa\mathcal{P}(e \twoheadrightarrow t)$ by definition. For any $\pi CRWL$-proof Δ by $size(\Delta)$ we denote the number of rules of the calculus used.
We assume we start with a proof for $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \twoheadrightarrow t$ which fulfils the conditions granted by Lemma 14. We define the relation $\lessdot$ over pairs of $\pi CRWL$-proofs Δ_1, Δ_2, by $\Delta_1 \lessdot \Delta_2$ iff $sa\mathcal{P}(\Delta_1) \subset sa\mathcal{P}(\Delta_2)$ or $sa\mathcal{P}(\Delta_1) = sa\mathcal{P}(\Delta_2)$ and $size(\Delta_1) < size(\Delta_2)$. Then for any $\pi \in \Pi^{e \twoheadrightarrow t}$ if $\pi \not\equiv e \twoheadrightarrow t$ then $\Pi^{\pi} \lessdot \Pi^{e \twoheadrightarrow t}$. We proceed by induction over $\lessdot$ applied over $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \twoheadrightarrow t$, let us do a case distinction over the rule applied at the root of the proof:

B Then $t \equiv \perp$ and $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \rightarrow^{0} e$ for which $\perp \sqsubseteq |e|$ holds.
RR Then $e \equiv X \equiv t$ and $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \rightarrow^{0} e$ for which $X \sqsubseteq X \equiv |e|$ holds.
DC If $e \equiv c \in CS^0$ then $t \equiv c$ and so $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \rightarrow^{0} e$ for which $c \sqsubseteq c \equiv |e|$ holds. Otherwise $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \equiv c(e_1, \ldots, e_n) \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t$. As we saw before for any $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e_i \twoheadrightarrow t_i$ we have $\Delta_{e_i \twoheadrightarrow t_i} \lessdot \Delta_{e \twoheadrightarrow t}$, hence by IH $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e_i \rightarrow^{*} e'_i$ such that $t_i \sqsubseteq |e'_i|$. But then $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \equiv c(e_1, \ldots, e_n) \twoheadrightarrow c(e'_1, \ldots, e'_n)$ and $c(t_1, \ldots, t_n) \sqsubseteq c(e'_1, \ldots, e'_n) \equiv |c(e'_1, \ldots, e'_n)|$.
POR Then we have $f(\overline{e}) \twoheadrightarrow t$. In case $t \equiv \perp$ lemma holds trivially for $f(\overline{e}) \rightarrow^{0} f(\overline{e})$ with $t \equiv \perp \sqsubseteq \perp \equiv |f(\overline{e})|$. Otherwise we proceed by a case distinct over f.
If $f \in FS^{\mathcal{P}}$ first we will see that

$$r(?\{\theta_{11}, \ldots, \theta_{1m_1}\} \uplus \ldots \uplus ?\{\theta_{n1}, \ldots, \theta_{nm_n}\}) \equiv r[\overline{X_{ij}/\theta_{i1}(X_{ij}) \ ? \ \ldots \ ? \ \theta_{im_i}(X_{ij})}]$$

with $i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}$ (remember that in the transformation of Definition 1 we had $\forall p_i \in \{p_1, \ldots, p_n\}$, $var(p_i) \cap var(r) = \{X_{i1}, \ldots, X_{ik_i}\}$). This is true because for any $Y \in var(r)$ as $var(r) \subseteq var(\overline{p})$ there must exists some $p_i \in \overline{p}$ such that $Y \in var(p_i)$, hence:

$$\begin{array}{ll} & Y(?\{\theta_{11}, \ldots, \theta_{1m_1}\} \uplus \ldots \uplus ?\{\theta_{n1}, \ldots, \theta_{nm_n}\}) \\ \equiv & Y(?\{\theta_{i1}, \ldots, \theta_{im_i}\}) \qquad (1) \\ \equiv & \theta_{i1}(Y) \ ? \ \ldots \ ? \ \theta_{im_i}(Y) \qquad (2) \\ \equiv & Y[Y/\theta_{i1}(Y) \ ? \ \ldots \ ? \ \theta_{im_i}(Y)] \\ \equiv & Y[\overline{X_{ij}/\theta_{i1}(X_{ij}) \ ? \ \ldots \ ? \ \theta_{im_i}(X_{ij})}] \qquad (3) \end{array}$$

(1) because $\forall i, j \ dom(\theta_{ij}) = var(p_i)$ and $\overline{p}$ is linear; (2) because $Y \in var(p_i) = dom(\theta_{ij})$ for any $\theta_{ij} \in \{\theta_{i1}, \ldots, \theta_{im_i}\}$; because $Y \in var(r)$ and $Y \in var(p_i)$, therefore $Y \in \{X_{i1}, \ldots, X_{ik_i}\}$ by definition.

Now let us do a preliminary version for $f \in FS^1$. Assume the rule used was $(f(p) \rightarrow r) \in \mathcal{P}$, then its transformation was $f(Y) \rightarrow if \ match(Y) \ then \ r[\overline{X_i/project_i(Y)}]$, where $\overline{X_i} = var(p) \cap var(r)$ and the rules for $match$ and each $project_i$ are $match(p) \rightarrow true$, $project_i(p) \rightarrow X_i$, and the proof must be of the following shape:

$$\frac{\begin{array}{l} e \twoheadrightarrow p\theta_1 \\ \ldots \\ e \twoheadrightarrow p\theta_m \quad r?\{\theta_1, \ldots, \theta_m\} \equiv r[\overline{X_i/\theta_1(X_i) \ ? \ \ldots \ ? \ \theta_m(X_i)}] \twoheadrightarrow t \end{array}}{\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} f(e) \twoheadrightarrow t} \ POR$$

Then for any $\theta_j \in \{\theta_1, \ldots, \theta_m\}$ we have $\Delta_{e \twoheadrightarrow p\theta_j} \lessdot \Delta_{f(e) \twoheadrightarrow t}$, so we choose arbitrary one of these θ_j and apply the IH to $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \twoheadrightarrow p\theta_j$ to get $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \to^* e'_j$ for some $e'_j \in Exp$ such that $p\theta_j \sqsubseteq |e'_j|$. But p is linear because it is in the left hand side of a rule, hence by Lemma 13 there must exist some $\sigma_j \in Subst$ such that $p\sigma_j \equiv e'_j$, and we can do:

$$\begin{array}{l} \hat{\mathcal{P}} \uplus \mathcal{M} \vdash f(e) \to if\ match(e)\ then\ r[\overline{X_i/project_i(e)}] \\ \to^* if\ match(e'_j)\ then\ r[\overline{X_i/project_i(e)}] \\ \equiv if\ match(p\sigma_j)\ then\ r[\overline{X_i/project_i(e)}] \\ \to if\ true\ then\ r[\overline{X_i/project_i(e)}] \to r[\overline{X_i/project_i(e)}] \end{array}$$

By Lemma 14 we know that $f(e) \twoheadrightarrow t$ is not a premise of itself, but then $sa\mathcal{P}(f(e) \twoheadrightarrow t) = (f(e) \twoheadrightarrow t) \uplus \mathcal{S}$, where $\mathcal{S} = \{\bigcup_{j \in \{1,\ldots,m\}} sa\mathcal{P}(e \twoheadrightarrow p\theta_j)\} \cup sa\mathcal{P}(r[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \twoheadrightarrow t)$. Now we will see that there is a proof of $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} r[\overline{X_i/project_i(e)}] \twoheadrightarrow t$ such that for any $\pi \in \Pi^{\Delta}$, $sa\mathcal{P}(\pi) \subseteq \mathcal{S}$. If $t \equiv \bot$ the proof is trivial using no program rule, thus $sa\mathcal{P}(\pi) = \emptyset$. Otherwise we proceed by induction on the structure of r. If $r \equiv Y \in \mathcal{V}$ such that $Y \notin \overline{X_i}$ then the proof is trivial because then $r[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \equiv Y \equiv r[\overline{X_i/project_i(e)}]$, and all the proofs starting from Y use no program rule and thus $sa\mathcal{P}(\pi) = \emptyset$. If $r \equiv X_i \in \overline{X_i}$ then as $r[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \twoheadrightarrow t$ then there must exist some $\theta_j \in \{\theta_1, \ldots, \theta_m\}$ such that $\theta_j(X_i) \twoheadrightarrow t$. But then we can do:

$$\frac{e \twoheadrightarrow p\theta_j \ \ X_i\theta_j \twoheadrightarrow t}{\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} r[\overline{X_i/project_i(e)}] \equiv project_i(e) \twoheadrightarrow t}\ POR$$

where $sa\mathcal{P}(X_i\theta_j \twoheadrightarrow s) = \emptyset$, as no program rule was used because $X_i\theta \in CTerm_{\bot}$, and $sa\mathcal{P}(e \twoheadrightarrow p\theta_j) \subseteq \mathcal{S}$: but then $sa\mathcal{P}(r[\overline{X_i/project_i(e)}] \twoheadrightarrow t) = sa\mathcal{P}(X_i\theta_j \twoheadrightarrow s) \cup sa\mathcal{P}(e \twoheadrightarrow p\theta_j) \subseteq \mathcal{S}$. If $r \in CS^0$ the proof is trivial as no program rule is involved. If $r \equiv c(a_1, \ldots, a_l)$ then $t \equiv c(t_1, \ldots, t_l)$ (as $t \not\equiv \bot$) and we can apply the HI over each $a_k[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \twoheadrightarrow t_k$ to get $a_k[\overline{X_i/project_i(e)}] \twoheadrightarrow t_k$ with $sa\mathcal{P}(a_k[\overline{X_i/project_i(e)}] \twoheadrightarrow t_k) \subseteq \mathcal{S}$, hence $r[\overline{X_i/project_i(e)}]$ by DC with

$$sa\mathcal{P}(r[\overline{X_i/project_i(e)}]) = \bigcup_k sa\mathcal{P}(a_k[\overline{X_i/project_i(e)}] \twoheadrightarrow t_k) \subseteq \mathcal{S}$$

If $r \equiv g(a_1, \ldots, a_l)$ with $g \in FS$, we can apply the IH to every $a_k[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \twoheadrightarrow s$ to get $a_k[\overline{X_i/project_i(e)}] \twoheadrightarrow s$ with $sa\mathcal{P}(a_k[\overline{X_i/project_i(e)}] \twoheadrightarrow s) \subseteq \mathcal{S}$. If the instance of the right hand side used in $g(a_1, \ldots, a_l)[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \twoheadrightarrow t$ was $r'\mu \twoheadrightarrow t$, then we can use the same instance for $g(a_1, \ldots, a_l)[\overline{X_i/project_i(e)}] \twoheadrightarrow t$, as we have reduced the arguments to the same values. Besides by definition $sa\mathcal{P}(r'\mu \twoheadrightarrow t) \subseteq sa\mathcal{P}(g(a_1, \ldots, a_l)[\overline{X_i/\theta_1(X_i)\ ?\ \ldots\ ?\ \theta_m(X_i)}] \twoheadrightarrow t) \subseteq \mathcal{S}$, hence $sa\mathcal{P}(g(a_1, \ldots, a_l)[\overline{X_i/project_i(e)}] \twoheadrightarrow t) \subseteq \mathcal{S}$.
But then as $sa\mathcal{P}(f(e) \twoheadrightarrow t) = (f(e) \twoheadrightarrow t) \uplus \mathcal{S}$ then $sa\mathcal{P}(r[\overline{X_i/project_i(e)}] \twoheadrightarrow t) \subseteq \mathcal{S} \subset sa\mathcal{P}(f(e) \twoheadrightarrow t)$ and we can apply the HI to get $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash r[\overline{X_i/project_i(e)}] \to^* e'$ such that $t \sqsubseteq |e'|$.

If $e \equiv match(e_1, \ldots, e_n)$ for some of these auxiliary functions, with rule $match(p_1, \ldots, p_n) \to true$, then as $t \not\equiv \bot$ we have $t \equiv true$ with:

$$\frac{e_1 \twoheadrightarrow p_1\theta_1 \quad \ldots \quad e_n \twoheadrightarrow p_n\theta_n \quad true \twoheadrightarrow true}{\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \equiv match(e_1, \ldots, e_n) \twoheadrightarrow true}\ POR$$

There could be more evaluations for each e_i but those are useless as $true$ is ground. Then we have $\Delta_{e_i \twoheadrightarrow p_i\theta_i} \lessdot \Delta_{e \twoheadrightarrow true}$ as we saw before, hence by IH $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e_i \to^* e'_i$ such that $p_i\theta_i \sqsubseteq |e'_i|$. But then by Lemma 13 there must exist some $\sigma_i \in Subst$ such that $p_i\sigma_i \equiv e'_i$, and we can do $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash match(e_1, \ldots, e_n) \to^* match(e'_1, \ldots, e'_n) \equiv match(p_1\sigma_1, \ldots, p_n\sigma_n) \to true$, and $true \sqsubseteq true \equiv |true|$.

If $e \equiv project(e_1)$ for some of these auxiliary functions, with rule $project(p) \to X$, then as $t \not\equiv \bot$ we have:

$$\frac{\begin{array}{l} e_1 \twoheadrightarrow p\theta_1 \\ \dots \\ e_1 \twoheadrightarrow p\theta_m \quad X(?\{\theta_1, \dots, \theta_m\}) \twoheadrightarrow t \end{array}}{\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \equiv project(e_1) \twoheadrightarrow t}\ POR$$

Then there must exist some $\theta_j \in \{\theta_1, \dots, \theta_m\}$ such that $X\theta_j \twoheadrightarrow t$, hence $t \sqsubseteq \theta_j(X)$. Besides we have $\Delta_{e_1 \twoheadrightarrow p\theta_j} \lessdot \Delta_{e \twoheadrightarrow t}$ as we saw before, hence by IH $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e_1 \to^* e'_j$ such that $p\theta_j \sqsubseteq |e'_j|$, to which we can apply Lemma 13 to get some $\sigma_j \in Subst$ such that $p\sigma_j \equiv e'_j$ and $\theta_j \sqsubseteq \sigma_j$. Therefore $t \sqsubseteq \theta_j(X) \sqsubseteq \sigma_j(X)$, so it is trivial to check that then $t \sqsubseteq |\sigma_j(X)|$ (by induction on the structure of t), and we can do $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash match(e_1) \to^* match(e'_j) \equiv match(p\sigma_j) \to \sigma_j(X)$.

If $e \equiv if\ e_1\ then\ e_2$ then as $t \not\equiv \bot$ we have:

$$\frac{\begin{array}{l} e_2 \twoheadrightarrow t_1 \\ \dots \\ e_1 \twoheadrightarrow true \quad e_2 \twoheadrightarrow t_m \quad t_1 ? \dots ? t_m \twoheadrightarrow t \end{array}}{\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \equiv if\ e_1\ then\ e_2 \twoheadrightarrow t}\ POR$$

There could be more evaluations for e_1 but those are useless as $true$ is ground. We have $\Delta_{e_1 \twoheadrightarrow true} \lessdot \Delta_{e \twoheadrightarrow t}$ as we saw before, hence by IH $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e_1 \to^* e'_1$ such that $true \sqsubseteq |e'_1|$. Then we can apply Lemma 13 to get some $\sigma_1 \in Subst$ such that $true \equiv true\sigma_1 \equiv e'_1$, so we can do $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash if\ e_1\ then\ e_2 \to^* if\ e'_1\ then\ e_2 \equiv if\ true\ then\ e_2 \to e_2$. Besides $t_1 ? \dots ? t_m \twoheadrightarrow t$ implies $t_j \twoheadrightarrow t$ for some $t_j \in \{t_1, \dots, t_m\}$ such that $t \sqsubseteq t_j$ and $\Delta_{e_2 \twoheadrightarrow t_j} \lessdot \Delta_{e \twoheadrightarrow t}$ as we saw before. We can apply the IH to get $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e_2 \to^* e'$ such that $t \sqsubseteq t_j \sqsubseteq |e'|$.

If $e \equiv e_1\ ?\ e_2$ then as $t \not\equiv \bot$ we have $e_i \twoheadrightarrow t$ for some $i \in \{1, 2\}$ with a proof to which we can apply the IH to get $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e_1\ ?\ e_2 \to e_i \to^* e'$ such that $t \sqsubseteq |e'|$.

Finally we are ready to prove Theorem 6 and Corollary 2.

Proof (For Theorem 6). Let $\hat{\mathcal{P}} \uplus \mathcal{M} = pST(\mathcal{P})$ be , where $\mathcal{M}$ is the set containing the rules for $\_?\_, if\_then\_$ and the new functions $match$ and $project$, and $\hat{\mathcal{P}}$ contains the new versions of the original rules of $\mathcal{P}$. If $e \in Exp, t \in CTerm_\bot$ are built using symbols on the signature of $\mathcal{P}$, then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ implies $\mathcal{P} \uplus \mathcal{M} \vdash_{\pi CRWL} e \twoheadrightarrow t$, which implies $\hat{\mathcal{P}} \uplus \mathcal{M} \vdash e \to^* e'$ such that $t \sqsubseteq |e'|$ by Lemma 15, that is, $pST(\mathcal{P}) \vdash e \to^* e'$.

Proof (For Corollary 2). Given some $t \in [\![e]\!]^{pl}_{\mathcal{P}}$ then by Theorem 6 exists some $e' \in Exp$ such that $pST(\mathcal{P}) \vdash e \to^* e'$ and $t \sqsubseteq |e'|$, hence $t \in [\![e]\!]^{rw}_{pST(\mathcal{P})}$ by definition. On the other hand if $t \in [\![e]\!]^{rw}_{pST(\mathcal{P})}$ then $t \in [\![e]\!]^{pl}_{pST(\mathcal{P})}$ by Corollary 1, but then $t \in [\![e]\!]^{pl}_{\mathcal{P}}$ by Theorem 5. For the second part, if $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ then $t \in [\![e]\!]^{pl}_{\mathcal{P}} = [\![e]\!]^{rw}_{pST(\mathcal{P})}$ by the first part, hence $\exists e' \in Exp$ such that $e \to^* e'$ and $t \sqsubseteq |e'|$. But as $t \in CTerm\ t$ is maximal wrt. $\sqsubseteq$ and so $t \equiv |e'|$ which implies $t \equiv e'$ (these are known properties of shells and $\sqsubseteq$). But then $pST(\mathcal{P}) \vdash e \to^* e' \equiv t$. If $pST(\mathcal{P}) \vdash e \to^* t$ then as $t \sqsubseteq t \equiv |t|$ then $t \in [\![e]\!]^{rw}_{pST(\mathcal{P})} = [\![e]\!]^{pl}_{\mathcal{P}}$: $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$.

8.1.10 A Flexible Framework for Programming with Non-deterministic Functions (Extended version)

A Flexible Framework for Programming with Non-deterministic Functions*

(Extended version)

Tech. Rep. SIC-9-08, 2008

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. The use of non-deterministic functions is a distinctive feature of modern functional logic languages. The semantics commonly adopted is *call-time choice*, a notion that at the operational level is related to the *sharing* mechanism of lazy evaluation in functional languages. However, there are situations where *run-time choice*, closer to ordinary rewriting, is more appropriate. In this paper we propose a unified formal framework where both semantics can co-exist for the same progam. This is done through a careful but neat combination of ordinary rewriting –to cope with run-time choice– with local bindings via a *let*-construct devised to express call-time choice. The result is a flexible framework into which existing call-time choice based languages can be embedded by means of a simple program transformation introducing *let*s in function definitions. We prove the adequacy of the embedding, as well as other relevant properties of the framework.

1 Introduction

Non-strict non-deterministic functions are a distinctive feature of modern functional logic languages (see [16] for a recent survey). It is known that the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions (see e.g. [26]). For term-rewriting based specifications, Hussmann [18] established a major distinction between *call-time choice* and *run-time choice*. Call-time choice is closely related to call-by-value and, in the case of strict semantics, it is easily implemented by innermost rewriting. In the case of non-strict semantics, things are more complicated, since the call-by-value view of call-time choice must include partial values. Operationally, this needs something similar to the sharing mechanism followed, by efficiency reasons, in (deterministic) functional languages under lazy evaluation. In contrast, run-time choice does not share, corresponds rather to call-by-name, and is realized by ordinary rewriting. For deterministic programs, run-time and call-time are able to produce the same set of values, but in general the set of values reachable by run-time choice is larger than that of call-time choice.

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), Promesas-CAM (S-0505/TIC/0407) and FAST-STAMP (TIN2008-06622-C03-01/TIN).

Non-deterministic functions with non-strict and call-time choice semantics were introduced in the functional logic setting with the *CRWL* framework [14, 15], in which programs are possibly non-confluent and non-terminating constructor-based term rewriting systems (*CTRS*). Since then, they are common part of daily programming in systems like Curry [17] or Toy [23]. Run-time choice has been rarely [2] considered as a valuable global alternative to call-time choice.

However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer's disposal. The purpose of this work is precisely proposing a clear, well-founded formal framework for doing that. The following example illustrates the interest of combining both semantics.

Example 1. Modeling grammar rules for string generation can be directly done by CTRS like the following (non-confluent and non-terminating) one, in which we assume that texts (terminals) are represented as strings (lists of characters), that can be concatenated with $++$ (defined in a standard way):

$$letter \to \text{"a"} \quad .... \quad letter \to \text{"z"} \qquad word \to \text{" "} \qquad word \to letter{+}{+}word$$

Disregarding syntax, that CTRS is a valid program in functional logic systems like Curry or Toy. Each individual reduction leads to a string. The generation of palindromes (of even length) could be done by the rewrite rules:

$$palindrome \to palAux(word) \qquad palAux(X) \to X \mathbin{++} reverse(X)$$

where *reverse* is defined in any standard way. It is important to remark that the definition of *palindrome/palAux* works fine only if call-time choice is adopted for non-determinism, meaning operationally that in the (partial) reduction

$$palindrome \to palAux(word) \to word \mathbin{++} reverse(word)$$

the two occurrences of *word* created by the rule of *palAux* must be shared. If run-time choice (i.e., ordinary rewriting) were used, the two occurrences of *word* could follow independent ways, and therefore *palindrome* could be reduced, for instance, to *"oops"*, which is not a palindrome. Two useful operators to structure grammar specifications are the alternative '|' and Kleene's '$*$' for repetitions:

$$X \mid Y \to X \qquad X \mid Y \to Y \qquad star(X) \to \text{" "} \qquad star(X) \to X{+}{+}star(X)$$

With them *letter* and *word* could be redefined as follows:

$$letter \to \text{"a"} \mid \text{"b"} \mid \ldots \mid \text{"z"} \qquad word \to star(letter)$$

The annoying fact is that this does not work! At least not under call-time choice, with which all the occurrences of *letter* created by *star* will be shared and therefore *word* will only generate words like *aaa* or *nnnn*, made with repetitions of the same letter. This problem was pointed out in [7], where a 'higher order trick' was suggested to overcome it. We discuss in depth that trick in Sect. 5. Notice, however, that it would be much simpler to consider that *star* follows run-time choice regime, so that the occurrences of *letter* created by *word* could evolve independently. We conclude that in this example neither call-time nor run-time choice are a good single option as semantics for the whole program. The definition of *palindrome* requires call-time, while *star* requires run-time.

To the best of our knowledge, no existing proposal offers the possibility of combining in the same program both kind of semantics. This paper addresses that problem at

a foundational level, deferring for the future the matters of implementing a concrete system or developing larger practical applications.

Our approach to combining run-time/call-time develops a natural idea: enhance run-time choice (i.e., ordinary rewriting) with a *let*-construction for local bindings to be governed by operational rules expressing the kind of sharing needed by call-time choice. We will call the enhanced rewriting relation *rt-let-rewriting*. In Example 1, the definitions of *letter*, *word* or *star* would remain the same. However, *palAux* would be encoded as *palAux(X)* $\rightarrow$ *let* $Y = X$ *in* Y *++ reverse(Y)* to ensure call-time choice for it. Or better, we can dispense with *palAux* and define directly *palindrome* $\rightarrow$ *let* Y $=$ *word in* Y *++ reverse(Y)*.

In spite of obvious general similarities, the use of *let*s in *rt-let*-rewriting must not be confused with other uses of local bindings in related scenarios, although of course some general similarities remain:
• *local definitions* of existing functional logic languages ([17, 23]): as in the functional case, they can be eliminated by lifting. Since those languages only support call-time choice, nothing really new is achieved with such *let*s, except program readability.
• *let*s of *lambda*-calculus with sharing ([6, 5]): they formalize sharing in lambda-calculus, but have nothing to do with non-determinism. Moreover, the underlying formalism is *lambda*-calculus instead of term rewriting.
• *let*s of *ct-let*-rewriting[1], proposed in [20] as a notion of one-step reduction adequately reflecting *CRWL*'s lazy call-time choice while avoiding the complexity of term graph rewriting [24, 11]. That use of *let*s follows a somehow complementary view to which is done here: in *ct-let*-rewriting, *let*s are introduced by the computation, even if the program does not contain *let*s at all, and must be combined with a restrictive function application rule that avoids the potential duplication of arguments caused by ordinary rewriting, in order to avoid run-time choice behavior. In contrast, in *rt-let*-rewriting function applications will be liberal (as ordinary rewriting is), and computations will not introduce *let*s, except those explicitly written in program rules to intentionally express call-time choice. As a consequence, the rules for function application and for *let*-management in [20] and in this paper are clearly different.

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about term rewriting systems. Section 3 introduces rewrite systems with *let*-bindings, presents the precise notion of *rt-let*-rewriting, and prove some of its properties. Section 4 shows that our new framework is a conservative extension of both pure run-time and pure call-time choice. The former will be almost obvious, while for the latter we will propose a program transformation $\mathcal{P} \mapsto \tau(\mathcal{P})$ introducing the necessary *let*s in $\mathcal{P}$, so that the behavior of $\mathcal{P}$ under call-time choice (as determined by the CRWL-semantics) and the behavior of $\tau(\mathcal{P})$ under run-time choice (as determined by *rt-let*-rewriting) coincide. In Section 5 we discuss in detail the question of wether our approach could be replaced by simpler ones; we point out some limits in the ability of run-time and call-time to simulate each other, and we show that our *rt-let*-rewriting compares advantageously to other alternative paths that might be followed. Finally Section 6 summarizes some conclusions. Fully detailed proofs, including many auxiliary results, can be found in [21].

[1] For the sake of clarity, we rename the '*let*-rewriting' relation of [20] to '*ct-let*-rewriting'.

2 Preliminaries

Constructor-based term rewrite systems

We consider a first order signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

Substitutions $\theta \in Subst$ are mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$.

A *constructor-based term rewriting system* $\mathcal{P}$ (*CTRS*, also called *program* along this paper) is a set of c-rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. Given a program $\mathcal{P}$, its associated rewrite relation $\to_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\theta] \to_{\mathcal{P}} \mathcal{C}[r\theta]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\theta \in Subst$. Notice that θ can instantiate extra variables to any expression. We write $\overset{*}{\to}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$.

Local bindings. The $CRWL_{let}$ framework

As explained in Section 1, in [20] we already considered local bindings in programs and expressions, but only for the purpose of characterizing call-time choice as a one-step reduction relation, *ct-let*-rewriting, that was proved to be equivalent to the semantics given by *CRWL*. As an auxiliary tool, we needed to extend the *CRWL* logic of [15] to the more general $CRWL_{let}$, a logic for call-time choice applicable to programs containing *let*s. In this section we briefly recall syntactic aspects of local bindings, as well as the $CRWL_{let}$ logic, that will be used later on. *Let-expressions* are defined as:

$$LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$$

where $X \in \mathcal{V}$, $h \in CS \cup FS$, and $e_1, \ldots, e_n \in LExp$. Recursive *let*s are not considered. In an expression $let\ X = e_1\ in\ e_2$, e_1 and e_2 are called the *defining* expression and the *body* of the *let*-expression, respectively. The notation $let\ \overline{X = a}\ in\ e$ abbreviates $let\ X_1 = a_1\ in\ \ldots\ in\ let\ X_n = a_n\ in\ e$. The notion of context is also extended to the new syntax: $\mathcal{C} ::= [] \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C} \mid h(\ldots, \mathcal{C}, \ldots)$.

From this point on, we assume that right-hand sides of program rules can contain *let*s, i.e., a program rule takes the form $f(\bar{t}) \to e$ with $e \in LExp$. However, we must stress the fact that in the $CRWL_{let}$ framework of [20] all programs, irrespective to the fact of using explicit *let*s or not, are to be assigned a call-time choice semantics. The main role of *let*s there is that they are introduced in *ct-let*-rewriting steps (see [20, 21] for the rules) precisely to ensure call-time choice. If *let*s are allowed in $CRWL_{let}$-programs is just for the sake of generality, but it is not difficult to realize that, within $CRWL_{let}$, any program using explicit *let*s has a semantically equivalent one with no *let*s at all.

The sets $FV(e)$ and $BV(e)$ of *free* and *bound* variables of $e \in LExp$ are defined as usual (see [20]). We assume a variable convention according to which the same variable symbol does not occur free and bound within an expression. Moreover, we assume that whenever θ is applied to $e \in LExp$, the necessary renamings of bound variables are done in e to ensure that $BV(e) \cap (dom(\theta) \cup vran(\theta)) = \emptyset$. These conditions avoid variable capture when applying a substitution, which can be then defined by the rules:

$$X\theta = \theta(X) \quad h(\overline{e})\theta = h(\overline{e\theta}) \quad (let\ X = e_1\ in\ e_2)\theta = (let\ X = e_1\theta\ in\ e_2\theta)$$

Free variables of contexts are defined as for expressions, so that $FV(\mathcal{C}) = FV(\mathcal{C}[\bot])$ $(= FV(\mathcal{C}[a])$, for any constant $a)$. However, the set $BV(\mathcal{C})$ of bound variables of a context is defined quite differently because it consists only of those *let*-bound variables visible from the hole of $\mathcal{C}$. Formally $BV([\ ]) = \emptyset$, $BV(h(\ldots,\mathcal{C},\ldots)) = BV(\mathcal{C})$, $BV(let\ X = e\ in\ \mathcal{C}) = \{X\} \cup BV(\mathcal{C})$, $BV(let\ X = \mathcal{C}\ in\ e) = BV(\mathcal{C})$. We will also employ the notion of *c-contexts*, which are contexts whose holes appear only within a nested application of constructor symbols, that is, $\mathcal{C} ::= [\ ] \mid c(e_1,\ldots,\mathcal{C},\ldots,e_n)$, with $c \in CS^n$, $e_1,\ldots,e_n \in LExp$.

As usual with non-strict languages, in order to express the semantics of expressions and programs the signature is enhanced with a new constant constructor symbol $\bot$, to represent the undefined value. Each syntactic domain $\mathcal{D} \in \{Subst, CSubst, Exp, LExp\}$ can be enlarged to the corresponding $\mathcal{D}_\bot$ of partial substitutions, etc. Notice, however, that $\bot$ does not appear in programs, nor it is introduced by any of the rewriting relations considered in the paper. Expressions in $LExp_\bot$ are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in LExp_\bot, \mathcal{C} \in Cntxt$.

The $CRWL_{let}$-logic defines the derivability relation $\mathcal{P} \vdash e \twoheadrightarrow t$, where $\mathcal{P}$ is a program, $e \in LExp_\bot$, $t \in CTerm_\bot$, indicating that t is an $\sqsubseteq$-approximation to a possible value for e, calculated with $\mathcal{P}$ according to call-time choice semantics. The inference rules defining this relation can be found in [20, 21].

Given a program $\mathcal{P}$, the approximated values of an expression $e \in LExp_\bot$ are collected in its $CRWL_{let}$-denotation $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash e \twoheadrightarrow t\}$. The *hypersemantics* gives a more active role to variables in the expression; it is the function $[\![e]\!]^{\mathcal{P}} : CSubst_\bot \to \mathcal{P}(CTerm_\bot)$ defined by $[\![e]\!]^{\mathcal{P}}\theta = [\![e\theta]\!]^{\mathcal{P}}$. The mention to $\mathcal{P}$ is frequently omitted. Semantics of expressions can be ordered by set inclusion, and hypersemantics are ordered by $[\![e]\!] \Subset [\![e']\!] \Leftrightarrow \forall\theta.[\![e]\!]\theta \subseteq [\![e']\!]\theta$ $(\Leftrightarrow \forall\theta.[\![e\theta]\!] \subseteq [\![e'\theta]\!])$.

The *shell* $|e|$ of an expression $e \in LExp_\bot$ is a partial c-term representing the outer constructed part (maybe implicit in *let*-bindings) of the expression, that is, the information that cannot disappear by reduction. Its formal definition is:

$$\begin{array}{llll} |X| & = X & |c(e_1,\ldots,e_n)| & = c(|e_1|,\ldots,|e_n|) \\ |f(e_1,\ldots,e_n)| & = \bot & |let\ X = e_1\ in\ e_2| & = |e_2|[X/|e_1|] \end{array}$$

Shells verify $|e| \in [\![e]\!]$, for any program and $e \in LExp_\bot$.

The *ct-let*-rewriting relation $\rightarrow^{ct}$ is proposed in [20] (where it was written $\rightarrow^{l}$) as a one-step rewriting relation corresponding to the $CRWL_{let}$-semantics. The rules governing $\rightarrow^{ct}$ can be found in [20, 21]. The main result in [20] is the equivalence of $CRWL_{\text{let}}$-derivability and $\rightarrow^{ct}$-reachability:

Theorem 1 ([20]). *$e \rightarrow^{ct^*}_{\mathcal{P}} t \Leftrightarrow \mathcal{P} \vdash e \twoheadrightarrow t$ ($\Leftrightarrow t \in [\![e]\!]^{\mathcal{P}}$), for any $\mathcal{P}, e \in LExp, t \in CTerm$.*

3 Run time choice with local bindings

We present here our framework for run-time choice with *let*-bindings. Syntactically, the family of programs is the same of $CRWL_{let}$, but the point of view changes completely, as argued in Section 1. In *rt-let*-rewriting –to be defined below– the reduction process does not create new *let*s, but only manages them conveniently. Therefore, writing explicit *let*s is essential in those points where the programmer wants a call-time choice behavior. Explicit *let*s in programs provide a great flexibility to the programmer, who can choose a specific behavior (shared/non shared) for each piece in a program rule. For instance, we could write a defining rule with the form $f(X, [Y|Ys]) \rightarrow$ *let* $U = X$ *in let* $V = Ys$ *in* e, indicating that the first argument of f and a part, but not the whole second one, are shared.

One of our concerns has been the careful treatment of extra variables in program rules, which is another point where call-time choice and run-time choice greatly differ. In call-time choice, the *CRWL*-semantics instantiates extra variables only with c-terms, but our *rt-let*-rewriting, which in particular attempts to be a strict generalization of ordinary rewriting (see Sect. 4), will instantiate them with any expression. This is a good point to recall that, as argued also in [20], rewriting (either ordinary, run-time or call-time rewriting) by itself is an ineffective operational procedure in presence of rules with extra variables, because a rewriting step using such rules requires a 'magic guessing' of an appropriate substitution for the extra variables. The natural solution to this problem is to perform *narrowing* instead of rewriting in such situations; that issue has been addressed for *ct-let*-rewriting in [19, 22], but for the case of *rt-let*-rewriting we postpone it for future work. Nevertheless, to have a rewriting notion is important, since typically the narrowing rules are designed to lift rewriting reductions.

Now we will define the run-time rewriting relation with local bindings (or *rt-let*-rewriting), written $\rightarrow^{rt}$ (or $\rightarrow^{rt}_{\mathcal{P}}$ if the program $\mathcal{P}$ is made explicit). To do this we will first define an auxiliary relation $\rightarrow^{rt'}$ for rewriting steps at the root position of an expression; a $\rightarrow^{rt}$-step is then defined as a $\rightarrow^{rt'}$-step put in context and fulfilling some additional conditions. In the following definition $\mathcal{P}$ is a program, $X, Y, Z \in \mathcal{V}$, $f \in FS, h \in FS \cup CS, t \in CTerm, e, e_i, a \in LExpr$, and $\mathcal{C}, \mathcal{C}' \in Cntx$.

Definition 1 (Run-time let rewriting relation $\rightarrow^{rt}$). *The auxiliary relation $\rightarrow^{rt'}$ is defined by the following rules:*

(Fapp) $f(\bar{t})\sigma \rightarrow^{rt'} e\sigma$ *if* $f(\bar{t}) \rightarrow e$ *is a rule of* $\mathcal{P}$, $\sigma \in LSubst$

(RBind) $let\ X = t\ in\ e \rightarrow^{rt'} e[X/t]$

(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^{rt'} e_2$ *if* $X \notin FV(e_2)$

(Flat$_1$) $h(\ldots, let\ X = e_1\ in\ e_2, \ldots) \rightarrow^{rt'} let\ X = e_1\ in\ h(\ldots, e_2, \ldots)$

(Flat$_2$) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{rt'} let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$

(LetIn) $let\ X = \mathcal{C}[e]\ in\ e' \rightarrow^{rt'} let\ Y = e\ in\ let\ X = \mathcal{C}[Y]\ in\ e'$
where Y *is fresh, if* $\mathcal{C} \neq [\,]$ *is a c-context and* $e \equiv f(\bar{e})$ *or* $e \in \mathcal{V}$.

Now, for any $\mathcal{C} \in Cntx$ we define $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$, if $e \rightarrow^{rt'} e'$ using any of the previous rules, and the following conditions hold, depending on the form of $e \rightarrow^{rt'} e'$:

i) If $e \rightarrow^{rt'} e'$ is $f(\bar{t})\sigma \rightarrow^{rt'} r\sigma$ by (Fapp) using $(f(\bar{t}) \rightarrow r) \in \mathcal{P}$ and $\sigma \in LSubst$, then $vran(\sigma|_{\setminus var(\bar{t})}) \cap BV(\mathcal{C}) = \emptyset$.

ii) If $e \rightarrow^{rt'} e'$ is $let\ X = t\ in\ a \rightarrow^{rt'} a[X/t]$ by (RBind), then $var(t) \subseteq BV(\mathcal{C})$.

iii) If $e \rightarrow^{rt'} e'$ is $let\ X = \mathcal{C}'[Y]\ in\ a \rightarrow^{rt'} let\ Z = Y\ in\ let\ X = \mathcal{C}'[Z]\ in\ a$ by (LetIn), then $Y \notin BV(\mathcal{C})$.

Some explanations about the rules follow. Rule **(Fapp)** allows to perform ordinary rewriting steps: when an expression matches the left-hand side of a program rule we can replace this expression with the right-hand side of the corresponding rule instance. Condition *i)* is imposed to avoid the capture of free extra variables introduced by σ. But we remark that in absence of extra variables in program rules, condition *i)* trivially holds and therefore (Fapp) (i.e., ordinary rewriting) can be done in any context.

The rest of the $\rightarrow^{rt}$-rules forget about the program and deal only with *let*-bindings. An important intuition is that if a step $e \rightarrow^{rt'} e'$ is performed using any of these rules that are independent from the program, then the set of $\rightarrow^{rt}$-reachable values (i.e. constructor terms) will be the same for e and e'. Therefore all non-determinism involved in these rules is *don't care*; only (Fapp) is *don't know*. Furthermore, we will see (Prop. 1) that those rules are not a source of non-termination. Let us now comment each of them.

When the defining expression of a *let*-binding has been reduced to a value then the rule **(RBind)** (restricted bind) can be used to propagate this value to the body of the *let*. The restriction expressed in condition *ii)* is needed to be coherent with the fact that in $\rightarrow^{rt}$ we use $LSubst$ for parameter passing, and so any variable can be potentially instantiated with a $LExp$. Now, notice that if we dropped condition *ii)*, a step like $let\ Y = X\ in\ (Y,Y) \rightarrow^{rt} (X,X)$ would be allowed; however, some of its particular cases (replacing the free variable X by concrete expressions) are not valid, as happens with $let\ Y = coin\ in\ (Y,Y) \rightarrow^{rt} (coin, coin)$, which is forbidden because it does not respect sharing. The property that any reduction step performed from an expression is also possible with any of its instances (obtained by a substitution of the kind allowed in parameter passing) is a desirable property, for it is very useful to reason about the programs. For example replacing the program rule $(f(X) \rightarrow let\ Y = X\ in\ (Y,Y))$ with $(f(X) \rightarrow (X,X))$ is unsound, because they provide different levels of sharing: this could be easily detected in our setting because the step $let\ Y = X\ in\ (Y,Y) \rightarrow^{rt} (X,X)$ is forbidden.

(Elim) erases a *let*-binding when the bound variable does not appear in the body.

The flattening rules **(Flat$_1$)** and **(Flat$_2$)** distribute the bindings to prevent derivations to become wrongly blocked. We remark that our variable convention ensures that application of (Flat$_1$) or (Flat$_2$) does not capture variables. The rule **(LetIn)** is designed to introduce *let*s only for expressions which are already shared, that is, which are present in a defining expression: introducing *let*s in more occasions would reduce the set of reachable values, causing incompleteness. Besides that, the context in which they appear must be a c-context because these (LetIn) steps are performed in order to enable a future (RBind) step, to propagate the partial value for the defining expression computed so far; the condition $C \neq [\,]$ avoids successive and useless applications of these rules. Specifically, the case $e \in \mathcal{V}$ in rule (LetIn) is needed to proceed in derivations blocked by the restrictions in (RBind), as illustrated by the program $\mathcal{P} = \{f(c(X)) \to true\}$ and the expression $let\ Y = c(X)\ in\ f(Y)$, to which (RBind) cannot be applied because X is free and therefore does not fulfil condition *ii)*. Without the case $e \in \mathcal{V}$ in (LetIn), that expression would be a normal form representing incorrectly a failed computation; but using (LetIn) as it is proposed we can do $let\ Y = c(X)\ in\ f(Y) \to^{rt} let\ Z = X\ in\ let\ Y = c(Z)\ in\ f(Y)$; now the computation can proceed successfully by applying (RBind,Fapp,Elim) yielding $let\ Z = X\ in\ f(c(Z)) \to^{rt} let\ Z = X\ in\ true \to^{rt} true$. The condition *iii)* affecting rule (LetIn) is only imposed to forbid useless steps of extraction of a bound variable, which are not needed to enable the application of (RBind).

As an example of derivation, consider the program $\mathcal{P} = \{coin \to 0, coin \to s(0), 0{+}X \to X, s(X){+}Y \to s(X{+}Y), double(X) \to let\ Y = X\ in\ Y{+}Y, pos(s(X)) \to true\}$ defining some easy operations for natural numbers (represented with 0 and s in the standard way). Notice the *let*-binding in the function *double*; it allows for example to evaluate $double(coin)$ to 0 or $s(s(0))$, but not to $s(0)$ (that could be obtained with $\to^{rt}$ if the binding were not present). The following is a possible $\to^{rt}$-derivation with $\mathcal{P}$ for the expression $pos(double(double(coin)))$. At each step, the redex is underlined and the applied $\to^{rt}$-rule is indicated on the right:

$$
\begin{array}{ll}
pos(\underline{double(double(coin))}) & (Fapp)\\
\to^{rt} \underline{pos(let\ Y = double(coin)\ in\ Y + Y)} & (Flat_1)\\
\to^{rt} let\ Y = \underline{double(coin)}\ in\ pos(Y + Y) & (Fapp)\\
\to^{rt} let\ Y = \underline{(let\ Z = coin\ in\ Z + Z)}\ in\ pos(Y + Y) & (Flat_2)\\
\to^{rt} let\ Z = \underline{coin}\ in\ let\ Y = Z + Z\ in\ pos(Y + Y) & (Fapp)\\
\to^{rt} \underline{let\ Z = s(0)\ in\ let\ Y = Z + Z\ in\ pos(Y + Y)} & (RBind)\\
\to^{rt} let\ Y = \underline{s(0) + s(0)}\ in\ pos(Y + Y) & (Fapp)\\
\to^{rt} \underline{let\ Y = s(0 + s(0))\ in\ pos(Y + Y)} & (LetIn)\\
\to^{rt} \underline{let\ V = 0 + s(0)\ in\ let\ Y = s(V)\ in\ pos(Y + Y)} & (RBind)\\
\to^{rt} let\ V = 0 + s(0)\ in\ \underline{pos(s(V) + s(V))} & (Fapp)\\
\to^{rt} let\ V = 0 + s(0)\ in\ \underline{pos(s(V + s(V)))} & (Fapp)\\
\to^{rt} \underline{let\ V = 0 + s(0)\ in\ true} & (Elim)\\
\to^{rt} \underline{true} &
\end{array}
$$

This is not the only possible derivation, nor the shortest one, but it illustrates some interesting aspects of the run-time rewriting relation. After the first use of (Fapp) we obtain a *let* construction inside a function call, that is extracted by (Flat$_1$). The applications of (Flat$_N$) or (LetIn) enable the application of (RBind) and, ultimately, of (Fapp). The last (Fapp) step shows how lazy evaluation works, without evaluating the inner '+'. The final step erases residual bindings and obtain the expected value.

A first interesting property that we have pursued in the design of the relation $\rightarrow^{rt}$ is that the program rules, to be applied through (Fapp), should be the only potential source of non-termination. The following result shows that this is indeed so.

Proposition 1. *The relation $\rightarrow^{rt}{}_{\setminus Fapp}$ defined by the rules of Def. 1 except (Fapp) is terminating.*

The next result reflects the fact that all rules except (Fapp) are syntactic transformations that preserve the outer constructed part of the expressions. This is in fact a first partial soundness result about the relation $\rightarrow^{rt}$.

Proposition 2. *For any $e, e' \in LExp$, if $e \rightarrow^{rt\,*} e'$ does not use (Fapp), then $|e| \equiv |e'|$.*

4 *Rt-let*-rewriting as a conservative extension

We have presented a (run-time choice) rewriting notion able to express sharing by means of an explicit *let* construction in program rules. The purpose of this section is to show with technical care that the resulting framework indeed generalizes pure run-time choice –as realized by ordinary rewriting– and pure call-time choice –as realized by the *CRWL* approach [15, 20]–.

The first statement – *rt-let*-rewriting generalizes ordinary rewriting – is fairly straighforward: if *let*s do not appear in a program $\mathcal{P}$, then every step of ordinary rewriting is a valid *rt-let*-rewriting step performed by the rule **(Fapp)** of Def. 1, because the absence of *let*s implies that $BV(\mathcal{C}) = \emptyset$ for any context $\mathcal{C}$, which guarantees the condition *i)* in Def. 1. Therefore, we have:

Theorem 2 (*Rt-let*-rewriting extends rewriting).
If $\mathcal{P}$ is a program with no lets, then $e \rightarrow_{\mathcal{P}} e' \Leftrightarrow e \rightarrow^{rt}_{\mathcal{P}} e'$, for any $e, e' \in Exp$.

To compare *rt-let*-rewriting with the *ct-let*-rewriting relation of [20] is more complicated since, despite their rough similarity, both relations are quite different; as a matter of fact, they are incomparable step by step. However, we will show how a program $\mathcal{P}$ can be transformed into another $\tau(\mathcal{P})$ that behaves, under *rt-let*-rewriting, as $\mathcal{P}$ with respect to *ct-let*-rewriting It is interesting to remark in advance that we will base the proof of adequacy of τ in semantic properties of $CRWL_{let}$, instead of reasoning directly about *ct-let*-rewriting derivations.

The transformation τ introduces *let*-bindings in the rules of a program in order to simulate call-time choice semantics, and is defined as follows:

Definition 2 (Sharing transformation τ). *Given a program rule $R \equiv f(\bar{t}) \rightarrow e$, its transformed rule is $\tau(R) \equiv (f(\bar{t}) \rightarrow let\ \overline{Y = X}\ in\ e[\overline{X/Y}])$ where $FV(e) = \overline{X}$ and $\overline{Y}$ is a linear tuple of fresh variables.*
The transformation is naturally extended to programs as $\tau(\mathcal{P}) = \{\tau(R) | R \in \mathcal{P}\}$.

This transformation introduces a *let*-binding for each variable in the right-hand side of a program rule. For example, for the program $\mathcal{P} = \{coin \rightarrow 0, coin \rightarrow 1, pair(X) \rightarrow (X, X)\}$ the last rule is transformed as $pair(X) \rightarrow let\ Y = X\ in\ (Y, Y)$, and if we evaluate now $pair(coin)$ we can obtain $(0, 0)$ and $(1, 1)$ but not $(0, 1)$ or $(1, 0)$; that reflects the evaluation of $pair(coin)$ with call-time choice.

The expected property of τ is that $\tau(\mathcal{P})$, if executed under *rt-let*-rewriting. behaves as $\mathcal{P}$, if executed under call-time choice (as given by $CRWL_{let}$). In other terms, τ serves to simulate call-time choice within run-time choice. To prove it we start by showing that τ is harmless when performed in a call-time choice ambient, i.e., τ preserves $CRWL_{let}$-(hyper)semantics:

Theorem 3 (Adequacy of τ under $CRWL_{let}$). *For any program $\mathcal{P}$ and $e \in LExp$ we have $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\tau(\mathcal{P})}$. In particular, $[e]^{\mathcal{P}} = [e]^{\tau(\mathcal{P})}$.*

We now address the *soundness* of τ as simulation of call-time choice: we show that $\tau(\mathcal{P})$, executed with *rt-let*-rewriting, does not produce new results when compared to $\mathcal{P}$ with call-time choice. To that purpose, the basic technical result is the following one, stating that at each step $e \to^{rt} e'$ done with $\tau(\mathcal{P})$, the hypersemantics of the reduced expression e does not grow (it might decrease due to non-determinism if (Fapp) was used for the step).

Lemma 1 (One-step hyper-soundness of $\to^{rt}$ for $\tau(\mathcal{P})$). *For any program $\mathcal{P}$, $e, e' \in LExp$, if $e \to^{rt}_{\tau(\mathcal{P})} e'$ then $[\![e']\!]^{\tau(\mathcal{P})} \Subset [\![e]\!]^{\tau(\mathcal{P})}$.*

As a consequence, chaining several $\to^{rt}$-steps and taking into account that $[\![e]\!] \Subset [\![e']\!]$ implies $[e] \subseteq [e']$, we obtain the following:

Theorem 4. *For any program $\mathcal{P}$, $e, e' \in LExp$, $t \in CTerm$:*

a) $e \to^{rt\,*}_{\tau(\mathcal{P})} e'$ *implies* $[e']^{\tau(\mathcal{P})} \subseteq [e]^{\tau(\mathcal{P})}$
b) $e \to^{rt\,*}_{\tau(\mathcal{P})} t$ *implies* $e \to^{ct\,*}_{\tau(\mathcal{P})} t$

Part *b)*, which follows from *a)* and the equivalence of $\to^{ct}$ and the $CRWL_{let}$ semantics (Th. 1), establishes already a close relationship between $\to^{rt}$ and $\to^{ct}$, but it is not yet our final soundness result, because it mentions only the transformed program $\tau(\mathcal{P})$. With the aid of Theorem 3, it is now straightforward to formulate our desired soundness result:

Theorem 5 (Soundness of τ as simulation of call-time choice). *For any program $\mathcal{P}$, $e \in LExp$, $t \in CTerm$ we have that $e \to^{rt\,*}_{\tau(\mathcal{P})} t$ implies $e \to^{ct\,*}_{\mathcal{P}} t$.*

The next goal is proving *completeness* of the simulation, i.e., the reciprocal of Th. 5. The technical key for it is the following result, ensuring that any value in the $CRWL_{let}$-semantics of an expression e can be covered by a $\to^{rt}$ derivation starting from e.

Lemma 2 (Completeness lemma for $\to^{rt}$). *For any program $\mathcal{P}$, $e \in LExp, t \in CTerm_{\bot}$, if $\mathcal{P} \vdash e \twoheadrightarrow t$ then $e \to^{rt\,*}_{\mathcal{P}} e'$ for some $e' \in LExp$ such that $t \sqsubseteq |e'|$.*

Notice that the lemma, being a completeness result, does not mention the transformed program, and therefore constitutes a formal proof of the intuitive fact that the $CRWL_{let}$-semantics, designed to express call-time choice, cannot give more results than the more liberal *rt-let*-rewriting, a result which is interesting in itself.

If we apply Lemma 2 to $t \in CTerm$ (i.e., t is total), then $t \sqsubseteq |e'|$ means $t = |e'|$, which in particular implies that there is no function application in $|e'|$. One could

expect then that the *let*-bindings that could remain in e' could be eliminated by some $\rightarrow^{rt}$-steps, and therefore that for t total $\mathcal{P} \vdash e \twoheadrightarrow t$ implies $e \rightarrow^{rt\,*}_{\mathcal{P}} t$. However, this cannot be guaranteed for total but not ground t, because a variable X in t, which is free, can appear in e' inside a *let*-binding *let* $Y = X$ *in* ... that cannot be dropped off because of the condition *i)* imposed to $\rightarrow^{rt}$ in Def. 1. What can be proved is the following:

Theorem 6 (Completeness of $\rightarrow^{rt}$ wrt $CRWL_{let}$).
For any program $\mathcal{P}$, $e \in LExp$, and $t \in CTerm$, if $\mathcal{P} \vdash e \twoheadrightarrow t$, then:
a) $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}$ let $\overline{Y = X}$ in t', for some $t' \in CTerm$ such that $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$.
b) If in addition t is ground, then $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$.

Joining all these completeness results with the previous soundness results, the equivalence of $\mathcal{P}$ and $\tau(\mathcal{P})$ wrt $CRWL_{let}$, and the equivalence of $CRWL_{let}$-semantics and *ct-let*-rewriting, it is not difficult now to obtain the adequacy (soundness + completeness) of the transformation τ to express call-time choice under an overall run-time choice regime.

Theorem 7 (Adequacy of the simulation of call-time-choice).
For any program $\mathcal{P}$, $e \in LExp$, $t \in CTerm_\bot$:

a) $\mathcal{P} \vdash e \twoheadrightarrow t \;\Leftrightarrow\; e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} e'$, for some $|e'| \sqsupseteq t$.

b) If t is total, then $\mathcal{P} \vdash e \twoheadrightarrow t \;\Leftrightarrow\; e \rightarrow^{ct\,*}_{\mathcal{P}} t \;\Leftrightarrow\; e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}$ let $\overline{Y = X}$ in t' for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$.

c) If t is total and ground, then $\mathcal{P} \vdash e \twoheadrightarrow t \Leftrightarrow e \rightarrow^{ct\,*}_{\mathcal{P}} t \Leftrightarrow e \rightarrow^{rt\,*}_{\tau(\mathcal{P})} t$

5 Discussion: could it be done simpler?

In this section we examine with some detail other possibilities to achieve the integration of run-time and call-time choice. First of all, we showed in [20] that no program transformation can perfectly mimic call-time choice within ordinary rewriting (i.e., within run-time choice without *let*s) due to their different closedness properties under substitutions. We show here that the opposite perfect imitation (run-time choice within call-time choice) is not possible either, in this case due to different compositionality properties of both kind of choices. We include the proof because of its remarkably simplicity, thanks to the equivalence of semantics and reduction for call-time choice ([20]) and the strength of some essential results about semantics.

Theorem 8. *There are programs $\mathcal{P}$ for which no program $\mathcal{P}'$ can verify the following property* (P)*: $e \rightarrow^{rt^*}_{\mathcal{P}} t \Leftrightarrow e \rightarrow^{ct^*}_{\mathcal{P}'} t$ for any ground $e \in Exp$, $t \in Cterm$.*

Proof. The following simple program suffices: $\mathcal{P} \equiv \{f(X) \rightarrow (X, X),\ coin \rightarrow 0,\ coin \rightarrow 1\}$. Assume there exists $\mathcal{P}'$ verifying *(P)*. Since $f(coin) \rightarrow^{rt^*}_{\mathcal{P}} (0, 1)$, *(P)* implies that $f(coin) \rightarrow^{ct^*}_{\mathcal{P}'} (0, 1)$ and then, because of the equivalence of *ct-let*-rewriting $\rightarrow^{ct^*}$ and *CRWL*-derivability (Th. 1), we have $\mathcal{P}' \vdash f(coin) \twoheadrightarrow (0, 1)$. By a compositionality property of call-time choice (see e.g. [22], Th. 1), there must be $t \in CTerm_\bot$ such that $\mathcal{P}' \vdash coin \twoheadrightarrow t$ and $\mathcal{P}' \vdash f(t) \twoheadrightarrow (0, 1)$. Now we distinguish some cases depending on the value of t (notice that t might be partial):

(a) If $t \equiv \bot$, then monotonicity of $CRWL$-derivability ([15]) proves that $\mathcal{P}' \vdash f(s) \twoheadrightarrow (0,1)$ for any $s \in CTerm_\bot$, in particular $\mathcal{P}' \vdash f(0) \twoheadrightarrow (0,1)$, and therefore $f(0) \rightarrow^{ct^*}_{\mathcal{P}'} (0,1)$, by Th. 1. Then, again by *(P)*, $f(0) \rightarrow^{rt^*}_{\mathcal{P}} (0,1)$, but this is not true.

(b) If $t \equiv 0$, then $\mathcal{P}' \vdash f(t) \twoheadrightarrow (0,1)$ leads to a contradiction as in (a). The cases $t \equiv 1$, $t \equiv Y$ or $t \equiv c(\overline{s})$ for a constructor c different from $0, 1$ lead to similar contradictions.

Some facts to be noticed: first, the program used in the proof is an ordinary CTRS (it does not use *let*s at all), and therefore the relation $\rightarrow^{rt}$ could by replaced by ordinary rewriting $\rightarrow$ (Th. 2) along Th. 8 and its proof. Second, the groundness restriction for e, t in the theorem is not a weakness, but quite the opposite (since the proposition as it is trivially implies the proposition dropping the groundness restriction). Third, the result is true even if transformed programs $\mathcal{P}'$ are allowed to be HO in the sense of [12], since the properties of *CRWL*-semantics used in the proof are also true for such HO extension.

Theorem 8 does not preclude the existence of other more sophisticated program transformations that, by changing the representation of expressions, could be suitable to express run-time choice within existing systems that use call-time choice (e.g., Curry [17] or Toy [23]). At a first sight, an old well-known HO technique [1] for delaying evaluation, based on the fact that partial applications are not evaluated, could help. We discuss it now with the aid of Example 1, where we encountered the problem of achieving run-time choice behavior for *star*. To clarify the discussion we use HO syntax and types (as existing systems do). The trick consists in replacing the original definitions of *String* generators like *letter,word,palindrome*, which had type *String* (an alias for *[Char]*), by a new functions of type *() → String* (here *()* plays the role of a *dummy* type). The type of *star* would be changed also to *star:: (() → String) → (() → String)*, and the program will be recoded as (we show only a part of it):

letter () → "a" .... letter () → "z" *word () → star letter ()*
star X () → "" *star X () → (X ()) ++ star X ()*

Now *letter* and *(star letter)* are partial applications, and *word ()* evaluates to *"ab"*, among other values, so that a run-time choice behavior for *star* has been achieved. This is a nice trick, used for parsing in [7, 8], but has some noticeable drawbacks and limitations, when compared to our approach:

(i) It requires to change the natural type of functions: moreover that change is global, and not localized in the functions for which one desires run-time choice behavior. If one wants generality and allows the inclusion of run-time functions at any point in the program, then the types of *all* functions f need to be artificially changed with dummy arguments, and thus the resulting code is much less natural.

(ii) An even more serious problem is that the trick is not general enough as to deal with matching. Consider, for instance, that we want a run-time choice regime for a function *f['a' | Xs]) → (Xs,Xs)*, so that *f(word)* can be reduced to *("a","b")*, among (infinitely many) other values. What type should be assigned to f in the HO-encoding? If we keep the 'original' type *f:: String → (String,String)*, then f cannot be applied directly to *word*; instead, we must consider *f (word ())*, but this can be reduced to *("a","a")* or *("b","b")* but not to *("a","b")*. Switching to the type *f::*

(() → String) → (String,String) does not solve the problem, because any suitable definition for f's needs to do some evaluation work with its argument (in order to match it with *['a' | X]*); but, at this point, what else can be done with an argument of type *() → String* except applying it to *()*, thus losing run-time choice behavior for f? Trying to overcome the problem we could think of a re-revision of all types, but it is fairly unclear how to do that, and anyhow it shows that a general technique to encode run-time choice in a host HO typed language following call-time choice can be rather cumbersome, if possible at all.
(iii) It requires to use HO to express FO run-time, thus mixing unnecessarily two concerns. Moreover, it is known (see e.g. [22]) that HO functions with call-time choice have subtle behaviors, so their use cannot be alleged to be free of surprises for the programmer.

In contrast to all this, our approach:
(i) seamlessly integrates types (the distinction run-time/call-time is irrelevant for types) and matching (nothing special must be done),
(ii) is more modular due to its local flavor (adopting call-time for a function affects only to its definition).
(iii) keeps the concerns FO/HO separated, and therefore could be more easily adapted to existing systems or frameworks that are directly based in FO rewriting (e.g., Maude [9]). The extension of the framework to HO can be addressed as an independent matter, realizable in standard ways followed in other works: adapting the theory to HO [12], adopting a FO translation [13, 4], or both [22]. In such a HO extension the management of call-time choice could be made even more modular and abstract through a HO polymorphic function *call_time F X → let Y = X in F Y*. With this function (that can be generalized to greater arities) we can get call-time versions of functions following other regimes,
(iv) last but not least, we give formal foundations to our approach, while nothing similar does exist for the HO-approach to simulation of run-time within call-time (and the question might be not trivial, as argued before).

6 Conclusions

We have proposed a new formal framework for (first order) programming with non-deterministic functions. The novelty is that, in contrast to existing languages where a decision is taken a priori about the semantics (run-time choice/call-time choice) of non-determinism adopted for functions, our approach allows using different semantics within the same program, which reveals itself as a very useful resource in many cases.This is achieved with great flexibility, because the selection of semantics can be done at the level of individual arguments or subexpressions, not only at the level of the complete definition of a function.

Our approach in a nutshell could be described as follows: to combine run-time choice and call-time choice, add a *let*-construct to a run-time choice framework (e.g., ordinary rewriting), and impose appropriate laws to the propagation of bindings contained in *let*s. Pure run-time choice (call-time choice resp.) is then achieved by not using *let*s at all (introducing *let*s for all function defining rules, resp.). Being the ideas so simple, two false impressions might arise: that existing frameworks are sufficient to cope with the combination of semantics, or that proving properties of the combined

framework is a routine task. Sect. 5 gets rid of the first illusion; regarding the second issue, it is interesting to observe that the proof of adequacy of our simulation of call-time choice (Sect. 4), besides of not being trivial, relies heavily on semantic properties of $CRWL_{let}$ (some of them new, see [21]), in a new strong evidence of the power, argued in [22], of using semantics to prove results about functional logic reductions.

The syntax presented here for our framework can be thought as a core syntax, that could be put in practice in different (mixable) ways:

(i) As syntactic sugar, each function can be declared as run-time or call-time, and its definition must be interpreted (and transformed, in the case of call-time) accordingly. A default declaration (call-time, most probably) could be assumed.

(ii) We can program using the core syntax, that is, with explicit *let*s. This gives a finer control, since we can choose specific behavior (shared/non shared) to each piece in an expression.

(iii) In a HO setting, the introduction of *let*s for call-time choice can be hidden in the function *call_time* $F\ X \to let\ Y = X\ in\ F\ Y$ introduced in Sect. 5. .

Having on hand simultaneously run-time choice and call-time choice (a non-sharing and a sharing procedure, respectively) is useful not only for programming purposes, but also for devising and justifying in a formal basis program transformations or implementation techniques. As an example consider the function *repeat*, programmed to follow call-time choice: *repeat(X)* → *let Y=X in [Y|repeat(Y)]*.
With this definition, an expression of the form *repeat(e)* reduces to the expression *let Y=e in [Y|repeat(Y)]*, and therefore recursive invocations to *repeat* (and there might be an arbitrarily large number of them in a lazy computation) generate successive *let*-bindings *let Z=Y in [Z|repeat(Z)]*, etc. However, intuitively only the first *let Y=e* is really needed, since then Y is already a shared value for which new sharings are useless. This suggests (automatically) replacing the original definition of *repeat* by an optimized variant *repeat(X)* → *let Y=X in [Y|repeat'(Y)]*, where the auxiliary *repeat'* is defined as *repeat'(X)* → *[X|repeat'(X)]*, thus avoiding the useless *let*s. We see some analogy between these *let*-binding savings described here and the implementation of sharing in some Curry systems [3] that try to avoid unnecessary creation of *suspensions*. A thorough investigation of these issues is left for future work. We simply remark here the potential applicability of our framework as a suitable formalism for making and proving precise statements.

We contemplate other relevant subjects of future work:

- The notion of rewriting given here should be lifted to a notion of narrowing, as was done in [19, 22] for the case of call-time choice.

- We must build an implementation of our framework. It should include types and HO functions, but we do not expect important novelties in this extension with respect to similar tasks performed in previous frameworks.

- We must invest some effort in producing a collection of program examples and programming patterns that make sensible use of the combination of run-time choice and call-time choice. From them, we should gain more insights about how, when and why making use of the combination.

References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.
2. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
3. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185, Nancy, France, March 2000. Springer LNCS 1794.
4. S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Fuji International Symposium on Functional and Logic Programming*, pages 335–353, 1999.
5. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.
6. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *POPL*, pages 233–246, 1995.
7. R. Caballero-Roldán and F. López-Fraguas. Parsing with non-deterministic functions. In *Proc. Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98*, pages 1–16, 1998.
8. R. Caballero-Roldán and F. López-Fraguas. A functional-logic perspective on parsing. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 85–99, London, UK, 1999. Springer-Verlag.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, pages 76–87. Springer LNCS 2706, 2003.
10. J. Dios-Castro and F. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science 188*, pages 3–19, 2007.
11. R. Echahed and J.-C. Janodet. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.
12. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.
13. J. C. González-Moreno. A correctness proof for Warren's ho into fo translation. In *GULP*, pages 569–584, 1993.
14. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.
15. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
16. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
17. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
18. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
19. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Narrowing for non-determinism with call-time choice semantics. In *Proc. Workshop on Logic Programming (WLP'07), Tech. Rep. 434 Univ. Wurzburg*, pages 224–233, 2007.

20. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming*, pages 197–208. ACM Press, 2007.
21. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions. http://gpd.sip.ucm.es/fraguas/papers/iclp08long.pdf, 2008.
22. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
23. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
24. D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.
25. J. Rodríguez-Hortalá. El indeterminismo en programacin lógico-funcional: un enfoque basado en reescritura. Trabajo de Investigación de Tercer Ciclo, Dpto. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Jun. 2007.
26. H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

A Proofs of the results

Figures 1 and 2 show the $CRWL_{let}$ calculus and *ct-let*-rewriting relation, respectively, defined in [20]. In Figure 3 we show an extended definition of the rule of the run-time let rewriting relation $\rightarrow^{rt}$ in which some conditions has been made explicit. This formulation is equivalent to the one in Def. 1 and is used intensively in the proofs.

$$\textbf{(B)}\ \frac{}{e \twoheadrightarrow \bot} \qquad \textbf{(RR)}\ \frac{}{X \twoheadrightarrow X} \quad X \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{e_1 \twoheadrightarrow t_1\ \ldots\ e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)} \quad c \in CS^n, t_i \in CTerm_\bot$$

$$\textbf{(OR)}\ \frac{e_1 \twoheadrightarrow t_1\theta\ \ldots\ e_n \twoheadrightarrow t_n\theta \quad e\theta \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t} \quad \begin{array}{l} f(\bar{t}) \rightarrow e \in \mathcal{P} \\ \theta \in CSubst_\bot \end{array}$$

$$\textbf{(Let)}\ \frac{e_1 \twoheadrightarrow t_1 \quad e[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e \twoheadrightarrow t}$$

Fig. 1. Rules of $CRWL_{let}$

For the derivations over $\rightarrow^{rt}$ sometimes we will do a derivation assuming that the rule was applied at the top of the expression, thus making a case distinction over the rule of $\rightarrow^{rt\,\prime}$, and then we will see how this result can be propagated to a rewriting step in any subexpression. We will call this latter step (Contx), which is just an application of $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$ if $e \rightarrow^{rt\,\prime} e'$, while the former cases are applications of $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$ if $e \rightarrow^{rt\,\prime} e'$ for $\mathcal{C} = []$. We will also use $\rightarrow^{rt}$ instead of $\rightarrow^{rt\,\prime}$ by an

(Contx)	$\mathcal{C}[e] \to_l \mathcal{C}[e']$, if $e \to_l e'$, $\mathcal{C} \in Cntxt$
(LetIn)	$h(\ldots,e,\ldots) \to_l let\ X = e\ in\ h(\ldots,X,\ldots)$ if $h \in CS \cup FS$, e takes one of the forms $e \equiv f(\overline{e'})$ with $f \in FS$ or $e \equiv let\ Y = e'\ in\ e''$, and X is a fresh variable
(Flat)	$let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \to_l let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$ assuming that Y does not appear free in e_3
(Bind)	$let\ X = t\ in\ e \to_l e[X/t]$, if $t \in CTerm$
(Elim)	$let\ X = e_1\ in\ e_2 \to_l e_2$, if X does not appear free in e_2
(Fapp)	$f(t_1\theta,\ldots,t_n\theta) \to_l e\theta$, if $f(t_1,\ldots,t_n) \to e \in \mathcal{P}$, $\theta \in CSubst$

Fig. 2. Rules of *ct-let*-rewriting

abuse of notation.

The following technical lemmas will be useful:

Lemma 3. *For any $\mathcal{C} \in Contx, e \in LExp_\bot$:*

i) $|\mathcal{C}[e]| \equiv |\mathcal{C}[|e|]|$
ii) $|e_1[X/e_2]| \equiv |e_1|[X/|e_2|]$

Proof (For lemma 3).

i) By the definition of shells.
ii) See [25].

Lemma 4. *For any $e \in Exp_\bot$, $t \in CTerm_\bot$ and program $\mathcal{P}$, if $\mathcal{P} \vdash e \twoheadrightarrow t$ then there is a derivation for $\mathcal{P} \vdash e \twoheadrightarrow t$ in which every free variable used belongs to $FV(e \twoheadrightarrow t)$.*

Proof (For lemma 4). A simple extension of the proof in [10].

Lemma 5. *For every $CRWL_{let}$ derivation $e \twoheadrightarrow t$ there exists $e' \in LExp_\bot$ which is syntactically equivalent to e module α-conversion, and a $CRWL_{let}$ derivation for $e' \twoheadrightarrow t$ such that if $\mathcal{B}$ is the set of bound variables used in $e' \twoheadrightarrow t$ and $\mathcal{E}$ is the set of free variables used in the instantiation of extra variables in $e' \twoheadrightarrow t$ then $\mathcal{B} \cap (\mathcal{E} \cup var(t)) = \emptyset$.*

Proof (For lemma 5). By lemma 4, if $\mathcal{F}$ is the set of free variables used in $e' \twoheadrightarrow t$, then $\mathcal{F} \subseteq FV(e' \twoheadrightarrow t)$, in fact $\mathcal{F} = FV(e' \twoheadrightarrow t)$, as $FV(e')$ and $FV(t)$ are used in the top derivation of the derivation tree for $e' \twoheadrightarrow t$. As by definition $\mathcal{E} \cup var(t) \subseteq \mathcal{F}$, if we prove $\mathcal{B} \cap \mathcal{F} = \emptyset$ then $\mathcal{B} \cap (\mathcal{E} \cup var(t)) = \emptyset$ is a trivial consequence. To prove that we will prove that for every $a \in LExp_\bot$ used in the derivation for $e' \twoheadrightarrow t$ we have $BV(a) \cap FV(a) = \emptyset$. We can build e' using α-conversion to ensure that $BV(e') \cap FV(e') = \emptyset$. This can be easily maintained as an invariant during the derivation, as the new *let* bindings that appear during the derivation are those introduced in the instances of the rule used during the **OR** steps, and be can ensure by α-conversion that $BV(a) \cap FV(a) = \emptyset$ for these instances too, as α-conversion leaves the hypersemantics untouched.

The auxiliary relation $\rightarrow^{rt\,\prime}$ is defined by the following rules:

(Fapp) $f(\overline{t})\sigma \rightarrow^{rt\,\prime} r\sigma$, if $(f(\overline{t}) \rightarrow r) \in \mathcal{P}$, $\sigma \in LSubst$
(RBind) $let\ X = t\ in\ e \rightarrow^{rt\,\prime} e[X/t]$, if $t \in CTerm$
(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^{rt\,\prime} e_2$, if $X \notin FV(e_2)$
(Flat$_1$) $h(\ldots, let\ X = e_1\ in\ e_2, \ldots) \rightarrow^{rt\,\prime} let\ X = e_1\ in\ h(\ldots, e_2, \ldots)$, with $h \in \Sigma$, if $X \notin FV(h(\ldots, [], \ldots))$
(Flat$_2$) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{rt\,\prime} let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin FV(e_3)$
(LetIn$_1$) $let\ X = \mathcal{C}[f(\overline{e})]\ in\ e \rightarrow^{rt\,\prime} let\ Y = f(\overline{e})\ in\ let\ X = \mathcal{C}[Y]\ in\ e$, with $f \in FS$, $Y \in \mathcal{V}$ fresh and $\mathcal{C} \neq []$ a c-context
(LetIn$_2$) $let\ X = \mathcal{C}[Y]\ in\ e \rightarrow^{rt\,\prime} let\ Z = Y\ in\ let\ X = \mathcal{C}[Z]\ in\ e$, with $Y, Z \in \mathcal{V}$, Z fresh and $\mathcal{C} \neq []$ a c-context

Now, for any $\mathcal{C} \in Cntx$ we have $\mathcal{C}[e] \rightarrow^{rt} \mathcal{C}[e']$, if $e \rightarrow^{rt\,\prime} e'$ using any of the previous rules, and in case $e \rightarrow^{rt\,\prime} e'$ is of the form:

i) $f(\overline{t})\sigma \rightarrow^{rt\,\prime} r\sigma$ by (Fapp) using $(f(\overline{t}) \rightarrow r) \in \mathcal{P}$ and $\sigma \in LSubst$, then $vran(\sigma|_{\setminus var(\overline{t})}) \cap BV(\mathcal{C}) = \emptyset$.
ii) $let\ X = t\ in\ a \rightarrow^{rt\,\prime} a[X/t]$ by (RBind), then $var(t) \subseteq BV(\mathcal{C})$.
iii) $let\ X = \mathcal{C}'[Y]\ in\ a \rightarrow^{rt\,\prime} let\ Z = Y\ in\ let\ X = \mathcal{C}'[Z]\ in\ a$ by (LetIn$_2$), then $Y \notin BV(\mathcal{C})$.

Fig. 3. Run-time let rewriting relation

Free and *bound* variables of $e \in LExp$ are defined as:

$$FV(X) = \{X\};\ FV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} FV(e_i);$$
$$FV(let\ X = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \backslash \{X\});$$
$$BV(X) = \emptyset;\ BV(h(\overline{e})) = \bigcup_{e_i \in \overline{e}} BV(e_i);$$
$$BV(let\ X = e_1\ in\ e_2) = BV(e_1) \cup BV(e_2) \cup \{X\}$$

We remark that the given definition of FV implies that recursive *let*-bindings simply do not exist. For instance, the binding in the expression $let\ X = s(X)\ in\ f(X)$ is not seen as recursive, despite of its aspect, because the occurrence of X in $s(X)$ is a free occurrence, and so it is 'different' from the bound X. Renaming the bound X to Y in the expression would give $let\ Y = s(X)\ in\ f(Y)$.

The following lemmas related to the sharing transformation $\tau()$ defined in Def. 2 will be useful later on. The first one states that $\tau()$ preserves free variables.

Lemma 6. *For any program rule $(l \rightarrow r)$ we have $FV(l \rightarrow r) = FV(\tau(l \rightarrow r))$, where $FV(f(\overline{p}) \rightarrow r)$ is defined as $var(\overline{p}) \cup FV(r)$.*

Proof (For lemma 6).

$$\begin{aligned}
&FV(\tau(l \rightarrow r)) = FV(l) \cup FV(let\ \overline{Y = X}\ in\ r[\overline{X/Y}]) = && \text{Def. of } \tau \\
&= FV(l) \cup \overline{X} \cup (FV(r[\overline{X/Y}]) \setminus \overline{Y}) && \\
&= FV(l) \cup \overline{X} \cup (\overline{Y} \setminus \overline{Y}) && \text{as } FV(r) = \overline{X} \\
&= FV(l) \cup \overline{X} \cup \emptyset = FV(l) \cup FV(r) = FV(l \rightarrow r) &&
\end{aligned}$$

The following result shows how we can introduce arbitrary *let*-bindings in expressions (for instance, those introduced by τ) while preserving their $CRWL_{let}$ semantics, and moreover their hypersemantics.

Lemma 7. *Let $\mathcal{P}$ be any program, $e \in LExp_\perp$ and $\overline{X}$ a linear n-tuple of arbitrary variables. Then*

$$[\![e]\!]^{\mathcal{P}} = [\![let\ \overline{Y = X}\ e[\overline{X/Y}]]\!]^{\mathcal{P}}$$

where $\overline{Y}$ is a linear n-tuple of fresh variables.

As a direct consequence, for any program $\mathcal{P}$ and $e \in LExp_\perp$ $[\![e]\!]^{\mathcal{P}} = [\![\tau(e)]\!]^{\mathcal{P}}$.

Proof (For lemma 7). Notice that as $\overline{X}$ and $\overline{Y}$ are linear tuples, the substitutions $[X_1/Y_1], \ldots [X_n/Y_n]$ can be reordered in any way obtaining the same substitution $[\overline{X/Y}]$. The notation $[\overline{X_{1..i}/Y_{1..i}}]$ stands for the substitution $[X_1/Y_1, \ldots, X_i/Y_i]$. The proof of the lemma proceed by induction on the number of variables in $\overline{X}$. The base case is trivial because there is no let and the induction step is $(i \Rightarrow i+1)$:

$$\begin{array}{l}
[\![let\ \overline{Y_{1..i+1} = X_{1..i+1}}\ in\ e[\overline{X_{1..i+1}/Y_{1..i+1}}]]\!] =_{(1)} \\
[\![let\ \overline{Y_{1..i} = X_{1..i}}\ in\ e[\overline{X_{1..i+1}/Y_{1..i+1}}][Y_{i+1}/X_{i+1}]]\!] =_{(2)} \\
[\![let\ \overline{Y_{1..i} = X_{1..i}}\ in\ e[\overline{X_{1..i}/Y_{1..i}}]]\!] =_{IH} \\
[\![e]\!]
\end{array}$$

The step (1) is justified because it is a step with **(Bind)**, that preserves the hyper-semantics of the expression (see [20]); and the step (2) is also sound because $e[\overline{X_{1..i+1}/Y_{1..i+1}}][Y_{i+1}/X_{i+1}] = e[\overline{X_{1..i}/Y_{1..i}}]$ as Y_{i+1} is fresh and does not appear in e.

In this lemma we see how applying $\tau()$ to a single program rule does not change the denotation of expressions, the key to prove Theorem 3.

Lemma 8. *Let $\mathcal{P}$ be a program, $(l \to r) \in \mathcal{P}$ and $\mathcal{P}' = (\mathcal{P} \setminus \{l \to r\}) \cup \{l \to \tau(r)\}$, then for any $e \in LExp_\perp$ we have $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$.*

Proof (For lemma 8). We must prove that for any $\sigma \in CSubst_\perp$ and $e \in LExp_\perp$:

$$\mathcal{P} \vdash e\sigma \twoheadrightarrow t \Leftrightarrow \mathcal{P}' \vdash e\sigma \twoheadrightarrow t$$

We proceed by induction on the size k of the derivation for $\mathcal{P} \vdash e\sigma \twoheadrightarrow t$:

- $k = 0$: the derivations with respect to $\mathcal{P}$ or $\mathcal{P}'$ are the same as they do not use any rule of the program.
- $k \Rightarrow k+1$: For proving the $(\Rightarrow)$ part, if the derivation $\mathcal{P} \vdash e\sigma \twoheadrightarrow t$ starts with a (DC) or (Let) step, the proof is a direct application of I.H., and similarly for the $(\Leftarrow)$ part. The most interesting case is when the derivation starts with a (OR) step using the rule $(l \to r) \equiv (f(\bar{t}) \to r)$. Then $e\sigma$ must be of the form $f(e_1, \ldots, e_n)$ and the derivation with respect to $\mathcal{P}$ is:

$$\frac{e_1 \twoheadrightarrow t_1\theta\ \ \ldots\ \ e_n \twoheadrightarrow t_n\theta \quad r\theta \twoheadrightarrow t}{f(e_1, \ldots, e_n) \twoheadrightarrow t}\ (OR)$$

using $\theta \in CSubst_\perp$ with $dom(\theta) = FV(f(\bar{t}) \to r)$. By I.H. we have all the derivations $\mathcal{P}' \vdash e_i \twoheadrightarrow t_i\theta$ and we must search for a derivation for $\mathcal{P}' \vdash \tau(r)\theta \twoheadrightarrow t$. For the last we have $\tau(r)\theta \equiv (let\ \overline{Y = X}\ in\ r[\overline{X/Y}])\theta$, where $\overline{X} = FV(r)$ and $\overline{Y}$ are fresh variables. Applying θ, this expression is equivalent to $let\ \overline{Y = X\theta}\ in\ r[\overline{X/Y}]\theta$, and as θ does not affect the variables $\overline{Y}$, this is also $let\ \overline{Y = X\theta}\ in\ r[\overline{X/Y}]$. For

this expression we can perform a derivation applying the rule (Let) once for each binding $Y_i = X_i$ reaching a derivation for $r[\overline{X/Y}][\overline{Y/X\theta}] \equiv r\theta \to t$, which can be done by I.H.
For ($\Leftarrow$) in the case of (OR) (using the program rule $f(\overline{l}) \to \tau(r)$), if $(let\ \overline{Y = X\theta}\ in\ r[\overline{X/Y}]) \twoheadrightarrow t$ then $X_i\theta \twoheadrightarrow s_i$ and $r[\overline{X/Y}][\overline{Y/s}] \twoheadrightarrow t$ ($\overline{X}$ and $\overline{Y}$ are linear by hypothesis), for some tuple of c-terms $\overline{s}$. But $X_i\theta \in CTerm_\perp$, and then $X_i\theta \twoheadrightarrow s_i$ implies $s_i \sqsubseteq X_i\theta$ and in fact $[\overline{X/s}] \sqsubseteq \theta$. Then $r[\overline{X/s}] = r[\overline{X/Y}][\overline{Y/s}] \twoheadrightarrow t$ (as $FV(r) = \overline{X}$) implies that the derivation $r\theta \twoheadrightarrow t$ can be done with smaller size.

Now we are ready to prove Theorem 3.

Proof (For Theorem 3). Given $\mathcal{P} = \{\rho_1, \ldots, \rho_n\}$ and $e \in Exp_\perp$ then

$$[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\{\rho_1,\ldots,\rho_n\}} = [\![e]\!]^{\{\tau(\rho_1),\ldots,\rho_n\}} = \ldots = [\![e]\!]^{\{\tau(\rho_1),\ldots,\tau(\rho_n)\}}$$

applying lemma 8 n times.

Now we will focus on proving lemma 1, to do this we firstly need the following technical results:

Lemma 9. *Let $t \in CTerm$ linear and $\sigma \in LSubst_\perp$ such that for any $X_i \in \overline{X} = var(t)$ we have $\mathcal{P} \vdash X_i\sigma \twoheadrightarrow s_i$ for some $s_i \in CTerm_\perp$. Then $\mathcal{P} \vdash t\sigma \twoheadrightarrow t[\overline{X_i/s_i}]$.*

Proof (For lemma 9). By induction on the structure of t:

Base cases
- $t \equiv X$: Then $\overline{X} = \{X\}$ and $\mathcal{P} \vdash_{CRWL_{let}} X\sigma \twoheadrightarrow s$, so $t\sigma \equiv X\sigma \twoheadrightarrow s \equiv X[X/s] \equiv t[\overline{X_i/s_i}]$
- $t \equiv c \in CS^0$: Then $\overline{X} = \emptyset$ and $t\sigma \equiv c\sigma \equiv c \twoheadrightarrow c \equiv c\epsilon \equiv t\epsilon \equiv t[\overline{X_i/s_i}]$

Inductive step $t \equiv c(t_1, \ldots, t_n)$: As t is linear then we assume $\overline{X} = \overline{X_1} \uplus \ldots \uplus \overline{X_n}$, where $\overline{X_j} = var(t_j)$. Now we can build:

$$\frac{\dfrac{IH}{t_1\sigma \twoheadrightarrow t_1[\overline{X_1/s_1}]} \quad \ldots \quad \dfrac{IH}{t_n\sigma \twoheadrightarrow t_n[\overline{X_n/s_n}]}}{t\sigma \equiv c(t_1\sigma, \ldots, t_n\sigma) \twoheadrightarrow c(t_1[\overline{X_1/s_1}], \ldots, t_n[\overline{X_n/s_n}])}\ DC$$

Note how, by the linearity of t, the premises corresponding to each t_j are independent and so the induction hypothesis can be applied independently too. Besides $c(t_1[\overline{X_1/s_1}], \ldots, t_n[\overline{X_n/s_n}]) \equiv c(t_1, \ldots, t_n)\ [\overline{X_1/s_1}, \ldots, \overline{X_n/s_n}]$ for the same reason.

Lemma 10 (Weak compositionality of $CRWL_{let}$). *For any $\mathcal{P}$ and $e \in LExp_\perp$: $[\![\mathcal{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathcal{C}[t]]\!]$, if $BV(\mathcal{C}) \cap FV(e) = \emptyset$. In particular, $[\![let\ X = e_1\ in\ e_2]\!] = \bigcup_{t \in [\![e_1]\!]} [\![e_2[X/t]]\!]$*

Proof (For lemma 10). It follows the same schema of weak compositionality of [22].

The notion of *hypersemantics of a context* and its associated compositionality result are powerful proving tools that we will use to prove lemma 1.

Definition 3 (Hypersemantics of a context). *Given $\mathcal{C} \in Contx$ its hypersemantics $[\![\mathcal{C}]\!]$ is a transformer of hypersemantics of expressions. Given $\varphi : CSubst_\perp \to \mathcal{P}(CTerm_\perp)$ and $\theta \in CSubst_\perp$, $[\![\mathcal{C}]\!]$ is defined by induction over the structure of $\mathcal{C}$:*

- $[\![[]]\!]\varphi = \varphi$
- $[\![h(e_1, \ldots, \mathcal{C}, \ldots, e_n)]\!]\varphi\theta = \bigcup_{t \in [\![\mathcal{C}]\!]\varphi\theta} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!]$
- $[\![let\ X = \mathcal{C}\ in\ e]\!]\varphi\theta = \bigcup_{t \in [\![\mathcal{C}]\!]\varphi\theta} [\![let\ X = t\ in\ e\theta]\!]$
- $[\![let\ X = e\ in\ \mathcal{C}]\!]\varphi\theta = \bigcup_{t \in [\![e]\!]\theta} [\![\mathcal{C}]\!]\varphi(\theta[X/t])$

With this notion we can prove the following abstract and powerful compositionaly result for hypersemantics, generalizing (and simplifying the aspect of) lemma 10, which was formulated in terms of semantics.

Lemma 11 (Compositionality of hypersemantics). $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}]\!][\![e]\!]$

This result implies that in any context one can replace any subexpression by another one having the same hypersemantics (and therefore also the same semantics) without changing the hypersemantics (hence the semantics) of the global expression.

Proof (For lemma 11). By induction over the structure of contexts.

Base case $\mathcal{C} = []$: Then $[\![\mathcal{C}[e]]\!] = [\![e]\!] = [\![[]]\!][\![e]\!] = [\![\mathcal{C}]\!][\![e]\!]$, as $[\![[]]\!]$ is the identity function, by definition.

Inductive step

- $\mathcal{C} = h(e_1, \ldots, \mathcal{C}', \ldots, e_n)$: Then

$$
\begin{array}{ll}
[\![\mathcal{C}]\!][\![e]\!] = \lambda\theta. \bigcup_{t \in [\![\mathcal{C}']\!][\![e]\!]\theta} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!] & \\
= \lambda\theta. \bigcup_{t \in [\![\mathcal{C}'[e]]\!]\theta} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!] & \text{by IH} \\
= \lambda\theta. \bigcup_{t \in [\![(\mathcal{C}'[e])\theta]\!]} [\![h(e_1\theta, \ldots, t, \ldots, e_n\theta)]\!] & \text{by definition} \\
= \lambda\theta.[\![h(e_1\theta, \ldots, (\mathcal{C}'[e])\theta, \ldots, e_n\theta)]\!] & \text{by lemma 10} \\
= \lambda\theta.[\![(\mathcal{C}[e])\theta]\!] = [\![\mathcal{C}[e]]\!] &
\end{array}
$$

- $\mathcal{C} = let\ X = \mathcal{C}'\ in\ s$: Then

$$
\begin{array}{ll}
[\![\mathcal{C}]\!][\![e]\!] = \lambda\theta. \bigcup_{t \in [\![\mathcal{C}']\!][\![e]\!]\theta} [\![let\ X = t\ in\ s\theta]\!] & \text{by definition} \\
= \lambda\theta. \bigcup_{t \in [\![\mathcal{C}']\!][\![e]\!]\theta} [\![s\theta[X/t]]\!] & \text{by rule (Bind) of } \to^{ct} \\
= \lambda\theta. \bigcup_{t \in [\![\mathcal{C}'[e]]\!]\theta} [\![s\theta[X/t]]\!] & \text{by IH} \\
= \lambda\theta. \bigcup_{t \in [\![(\mathcal{C}'[e])\theta]\!]} [\![s\theta[X/t]]\!] & \text{by definition} \\
= \lambda\theta.[\![let\ X = (\mathcal{C}'[e])\theta\ in\ s\theta]\!] & \text{by lemma 10} \\
= [\![\mathcal{C}[e]]\!] &
\end{array}
$$

– $\mathcal{C} = let\ X = s\ in\ \mathcal{C}'$: Then

$$\begin{array}{ll}
[\![\mathcal{C}]\!][\![e]\!] = \lambda\theta.\ \bigcup_{t\in[\![s]\!]\theta} [\![\mathcal{C}']\!][\![e]\!](\theta[X/t]) & \\
= \lambda\theta.\ \bigcup_{t\in[\![s]\!]\theta} [\![\mathcal{C}'[e]]\!](\theta[X/t]) & \text{by IH} \\
= \lambda\theta.\ \bigcup_{t\in[\![s]\!]\theta} [\![(\mathcal{C}'[e])(\theta[X/t])]\!] & \text{by definition} \\
= \lambda\theta.\ \bigcup_{t\in[\![s\theta]\!]} [\![(\mathcal{C}'[e])(\theta[X/t])]\!] & \text{by definition} \\
= \lambda\theta.\ \bigcup_{t\in[\![s\theta]\!]} [\![((\mathcal{C}'[e])\theta)[X/t]]\!] & \\
= \lambda\theta.[\![let\ X = s\theta\ in\ (\mathcal{C}'[e])\theta]\!] & \text{by lemma 10} \\
= [\![\mathcal{C}[e]]\!] &
\end{array}$$

The following lemma, combined with lemma 11, will be one of the keys to prove lemma 1.

Lemma 12. *If $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$ and $X \notin FV(\mathcal{C})$ then $[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!] = [\![let\ X = e_1\ in\ \mathcal{C}]\!]$*

Proof (For lemma 12). By induction on the structure of $\mathcal{C}$:

Base case $\mathcal{C} = []$: This case is trivial as then $\mathcal{C}[let\ X = e_1\ in\ []] \equiv let\ X = e_1\ in\ \mathcal{C}$
Inductive steps

• $\mathcal{C} = h(a_1, \ldots, \mathcal{C}', \ldots, a_n)$: Then

$$\begin{array}{ll}
[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!]\varphi\theta & \\
= [\![h(a_1, \ldots, \mathcal{C}'[let\ X = e_1\ in\ []], \ldots, a_n)]\!]\varphi\theta & \\
= \bigcup_{t\in[\![\mathcal{C}'[let\ X=e_1\ in\ []]]\!]\varphi\theta} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] & \\
=_{IH} \bigcup_{t\in[\![let\ X=e_1\ in\ \mathcal{C}']\!]\varphi\theta} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] & \\
= \bigcup_{t\in(\bigcup_{s\in[\![e_1]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[X/s]))} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] & \\
= \bigcup_{s\in[\![e_1]\!]\theta}\ \bigcup_{t\in[\![\mathcal{C}']\!]\varphi(\theta[X/s])} [\![h(a_1\theta, \ldots, t, \ldots, a_n\theta)]\!] & \\
=_{(1)} \bigcup_{s\in[\![e_1]\!]\theta}\ \bigcup_{t\in[\![\mathcal{C}']\!]\varphi(\theta[X/s])} [\![h(a_1\theta[X/s], \ldots, t, \ldots, a_n\theta[X/s])]\!] & \\
= \bigcup_{s\in[\![e_1]\!]\theta} ([\![h(a_1, \ldots, \mathcal{C}', \ldots, a_n)]\!]\varphi(\theta[X/s])) & \\
= [\![let\ X = e_1\ in\ h(a_1, \ldots, \mathcal{C}', \ldots, a_n)]\!]\varphi\theta & \\
= [\![let\ X = e_1\ in\ \mathcal{C}]\!]\varphi\theta &
\end{array}$$

where:

- (1) : As $FV(a_1) \cup \ldots \cup FV(a_n) \subseteq FV(h(a_1, \ldots, \mathcal{C}', \ldots, a_n)) = FV(\mathcal{C})$ and $X \notin FV(\mathcal{C})$ by hypothesis, then $\forall i,\ X \notin FV(a_i)$. Besides $X \notin vran(\theta)$ by the variable convention, thus $X \notin FV(a_i\theta)$ and $a_i\theta[X/s] \equiv a_i\theta$ for any i.

• $\mathcal{C} = let\ Y = \mathcal{C}'\ in\ s$: Then

$$
\begin{aligned}
&[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= [\![let\ Y = \mathcal{C}'[let\ X = e_1\ in\ []]\ in\ s]\!]\varphi\theta \\
&= \bigcup_{t \in [\![\mathcal{C}'[let\ X = e_1\ in\ []]]\!]\varphi\theta} [\![let\ Y = t\ in\ s\theta]\!] \\
&=_{IH} \bigcup_{t \in [\![let\ X = e_1\ in\ \mathcal{C}']\!]\varphi\theta} [\![let\ Y = t\ in\ s\theta]\!] \\
&= \bigcup_{t \in (\bigcup_{r \in [\![e_1]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[X/r]))} [\![let\ Y = t\ in\ s\theta]\!] \\
&= \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![\mathcal{C}']\!]\varphi(\theta[X/r])} [\![let\ Y = t\ in\ s\theta]\!] \\
&=_{(1)} \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![\mathcal{C}']\!]\varphi(\theta[X/r])} [\![let\ Y = t\ in\ s\theta[X/r]]\!] \\
&= \bigcup_{r \in [\![e_1]\!]\theta} ([\![let\ Y = \mathcal{C}'\ in\ s]\!]\varphi(\theta[X/r])) \\
&= [\![let\ X = e_1\ in\ let\ Y = \mathcal{C}'\ in\ s]\!]\varphi\theta \\
&= [\![let\ X = e_1\ in\ \mathcal{C}]\!]\varphi\theta
\end{aligned}
$$

where:

- (1) : As $FV(s) \subseteq FV(\mathcal{C}) \cup \{Y\}$, and because we may assume $X \neq Y$ by α-conversion, and we also have $X \notin FV(\mathcal{C})$ by hypothesis, then $X \notin FV(s)$. Besides $X \notin vran(\theta)$ by the variable convention, thus $X \notin FV(s\theta)$ and $s\theta[X/r] \equiv s\theta$.

• $\mathcal{C} = let\ Y = s\ in\ \mathcal{C}'$: Then

$$
\begin{aligned}
&[\![\mathcal{C}[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= [\![let\ Y = s\ in\ \mathcal{C}'[let\ X = e_1\ in\ []]]\!]\varphi\theta \\
&= \bigcup_{t \in [\![s]\!]\theta} [\![\mathcal{C}'[let\ X = e_1\ in\ []]]\!]\varphi(\theta[Y/t]) \\
&=_{IH} \bigcup_{t \in [\![s]\!]\theta} [\![let\ X = e_1\ in\ \mathcal{C}']\!]\varphi(\theta[Y/t]) \\
&= \bigcup_{t \in [\![s]\!]\theta} \bigcup_{r \in [\![e_1]\!](\theta[Y/t])} [\![\mathcal{C}']\!]\varphi(\theta[Y/t][X/r]) \\
&=_{(1)} \bigcup_{t \in [\![s]\!]\theta} \bigcup_{r \in [\![e_1]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[Y/t][X/r]) \\
&= \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![s]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[Y/t][X/r]) \\
&=_{(2)} \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![s]\!]\theta} [\![\mathcal{C}']\!]\varphi(\theta[X/r][Y/t]) \\
&=_{(3)} \bigcup_{r \in [\![e_1]\!]\theta} \bigcup_{t \in [\![s]\!](\theta[X/r])} [\![\mathcal{C}']\!]\varphi(\theta[X/r][Y/t]) \\
&= \bigcup_{r \in [\![e_1]\!]\theta} [\![let\ Y = s\ in\ \mathcal{C}']\!]\varphi(\theta[X/r]) \\
&= [\![let\ X = e_1\ in\ let\ Y = s\ in\ \mathcal{C}']\!]\varphi\theta \\
&= [\![let\ X = e_1\ in\ \mathcal{C}]\!]\varphi\theta
\end{aligned}
$$

where:

- (1) : As $Y \in BV(\mathcal{C})$ and $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$ by hypothesis, then $Y \notin FV(e_1)$. Besides $Y \notin vran(\theta)$ by the variable convention, thus $Y \notin FV(e_1\theta)$ and $e_1\theta[Y/t] \equiv e_1\theta$. But then we can chain $[\![e_1]\!](\theta[Y/t]) = [\![e_1\theta[Y/t]]\!] = [\![e_1\theta]\!] = [\![e_1]\!]\theta$.
- (2) : We may assume $X \neq Y$ by α-conversion, so $X \notin dom([Y/t])$; we may assume $X \notin var(t)$ by lemma 5, so $X \notin vran([Y/t])$. But then we can

apply the substitution lemma to get, for any $e \in LExp_\bot$, $e[X/r][Y/t] \equiv e[Y/t][X/r[Y/t]] \equiv e[Y/t][X/r]$, as $Y \notin var(r)$ by lemma 5 . Hence $[X/r][Y/t] = [Y/t][X/r]$.

- (3) : As $FV(s) \subseteq FV(\mathcal{C})$ and $X \notin FV(\mathcal{C})$ by hypothesis, then $X \notin FV(s)$. Besides $X \notin vran(\theta)$ by the variable convention, thus $X \notin FV(s\theta)$ and $s\theta[X/r] \equiv s\theta$.But then we can chain $[\![s]\!]\theta = [\![s\theta]\!] = [\![s\theta[X/r]]\!] = [\![s]\!](s\theta[X/r])$.

Now we are ready to prove lemma 1.

Proof (For lemma 1). By a case distinction. It is not a surprise that the most difficult step was (Fapp), as the essence of the transformation is concentrated in this step.

(Fapp) As we are working with the transformed program the rule used in this step must be of the shape $R = (f(\overline{p}) \to let\ \overline{Y = X}\ in\ r[\overline{X/Y}])$ such that $\overline{X} = FV(r)$, where $(f(\overline{p}) \to r)$ is the original rule. Assume the step was:

$$f(\overline{p})\sigma \to^{rt} (let\ \overline{Y = X}\ in\ r[\overline{X/Y}])\sigma$$

Without loss of generality we may assume $dom(\sigma) \subseteq FV(R) = FV(f(\overline{p}) \to r)$, as free variables are preserved by $\tau()$, as stated by lemma 6. Then $dom(\sigma) \cap \overline{Y} \subseteq FV(R) \cap \overline{Y} = \emptyset$, as the variables in $\overline{Y}$ are fresh wrt the variables in R, by definition of $\tau()$. Besides, $FV(r[\overline{X/Y}]) = \overline{Y}$, as $\overline{X} = FV(r)$, so $r[\overline{X/Y}]\sigma \equiv r[\overline{X/Y}]$. Hence we can reformulate the step as:

$$f(\overline{p})\sigma \to^{rt} let\ \overline{Y = X\sigma}\ in\ r[\overline{X/Y}]$$

– If $\overline{X} = \emptyset$ then R remains the same as in the original program, $R = (f(\overline{p}) \to r)$, and r is ground, so the step was $f(\overline{p})\sigma \to^{rt} r\sigma \equiv r$. Then given $\theta \in CSubst_\bot$ such that $\vdash_{CRWL_{let}} r\theta \twoheadrightarrow t$, as r is ground then $\vdash_{CRWL_{let}} r \equiv r\theta \twoheadrightarrow t$. Besides, given $\overline{Z} = var(\overline{p})$ it is easy to prove that $\forall \gamma \in LSubst_\bot$, $i \in \{1, \ldots, n\}$, it happens $\vdash_{CRWL_{let}} p_i\gamma \twoheadrightarrow p_i[\overline{Z/\ \bot}]$ (by induction on the structure of $CTerm$), and we can do:

$$\frac{\overline{p_1\sigma\theta \twoheadrightarrow p_1[\overline{Z/\bot}]}\ \ldots \overline{p_n\sigma\theta \twoheadrightarrow p_n[\overline{Z/\bot}]}\quad \overline{r[\overline{Z/\bot}] \equiv r \twoheadrightarrow t}}{f(\overline{p})\sigma\theta \twoheadrightarrow t}\ OR$$

using the instance $(f(\overline{p}) \to r)[\overline{Z/\ \bot}] \in [\mathcal{P}]_\bot$.

– If $\overline{X} \neq \emptyset$, given $\theta \in CSubst_\bot$ such that $\vdash_{CRWL_{let}} (let\ \overline{Y = X\sigma}\ in\ r[\overline{X/Y}])\theta \twoheadrightarrow t$, by the variable convention $\overline{Y} \cap dom(\theta) = \emptyset$, and besides $FV(r[\overline{X/Y}]) = \overline{Y}$, as $\overline{X} = FV(r)$, hence $r[\overline{X/Y}]\theta \equiv r[\overline{X/Y}]$ and the derivation was:

$$\frac{X_1\sigma\theta \twoheadrightarrow s_1\quad \overset{\ldots}{\overline{(let\ Y_2 = X_2\sigma\theta\ in\ \ldots\ in\ r[\overline{X/Y}][Y_1/s_1] \twoheadrightarrow t}}}{(let\ \overline{Y = X\sigma}\ in\ r[\overline{X/Y}])\theta \equiv let\ \overline{Y = X\sigma\theta}\ in\ r[\overline{X/Y}] \twoheadrightarrow t}\ Let$$

But as $\overline{X}$ is linear and so does $\overline{Y}$, every $Y_j \in \overline{Y}$ is different from every $X_i \in \overline{X}$, and $\overline{Y} \cap (vran(\sigma) \cup vran(\theta)) = \emptyset$ by the variable convention, then $\forall i, j\ X_i\sigma\theta[Y_j/s_j] \equiv X_i\sigma\theta$, and so $\vdash_{CRWL_{let}} let\ \overline{Y = X\sigma\theta}\ in\ r[\overline{X/Y}] \twoheadrightarrow t$ iff for every $X_i \in \overline{X}$ exists some $s_i \in CTerm_\bot$ such that $\vdash_{CRWL_{let}} X_i\sigma\theta \twoheadrightarrow s_i$, and

$\vdash_{CRWL_{let}} r[\overline{X/Y}][\overline{Y/s}] \twoheadrightarrow t$.
Besides, as $FV(r[\overline{X/Y}]) = \overline{Y}$, as $\overline{X} = FV(r)$, then $r[\overline{X/Y}]\ [\overline{Y/s}] \equiv r[\overline{X/s}]$, hence $\vdash_{CRWL_{let}} r[\overline{X/s}] \twoheadrightarrow t$ and we can do:

$$\dfrac{\dfrac{\forall X_i \in \overline{X} \cap var(\overline{p}),\ X_i\sigma\theta \twoheadrightarrow s_i \text{ + lemma 9}}{\overline{p}\sigma\theta \twoheadrightarrow \overline{p}([\overline{X/s}])|_{var(\overline{p})} \equiv \overline{p}[\overline{X/s}]} \ (*)}{f(\overline{p})\sigma\theta \twoheadrightarrow t}\ OR$$

where (*) is the derivation:

$$\dfrac{\dfrac{\dfrac{r[\overline{X/s}] \twoheadrightarrow t}{\text{lemma 7 + } t \in [\![r]\!][\overline{X/s}]}}{t \in [\![let\ \overline{Y = X}\ in\ r[\overline{X/Y}]]\!][\overline{X/s}]}}{(let\ \overline{Y = X}\ in\ r[\overline{X/Y}])[\overline{X/s}] \twoheadrightarrow t}$$

using the instance $(f(\overline{p}) \to let\ \overline{Y = X}\ in\ r[\overline{X/Y}])[\overline{X/s}]$ of $[\mathcal{P}]_\perp$.

(RBind) This is a particular case of the rule (Bind) of $\to^{ct}$, see proof in [20].

(Elim) This is a particular case of the rule (Elim) of $\to^{ct}$, see proof in [20].

(Flat$_1$) Let us define a new rule:

(Dist) $\mathcal{C}[let\ X = e_1\ in\ e_2] \to let\ X = e_1\ in\ \mathcal{C}[e_2]$ for every $\mathcal{C} \neq []$ such that $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$ and $X \notin FV(\mathcal{C})$

This rule introduces non-termination for every program, but it will be useful because it generalizes the let distribution rules (Flat$_1$) and (Flat$_2$). We will see that a (Dist) step leaves the hypersemantics untouched. But now, as $[\![\mathcal{C}[e]]\!] = [\![\mathcal{C}]\!][\![e]\!]$ by lemma 11, we can chain $[\![\mathcal{C}[let\ X = e_1\ in\ e_2]]\!] = [\![\mathcal{C}[let\ X = e_1\ in\ []]]\!][\![e_2]\!] = [\![let\ X = e_1\ in\ \mathcal{C}]\!][\![e_2]\!] = [\![let\ X = e_1\ in\ \mathcal{C}[e_2]]\!]$, applying lemma 12 in the third step. Now as every (Flat$_1$) step is a particular case of a (Dist) step then (Flat$_1$) leaves the hypersemantics untouched.

(Flat$_2$) As every (Flat$_2$) step is a particular case of a (Dist) step then (Flat$_2$) leaves the hypersemantics untouched.

(LetIn$_1$) Let us define a new rule:

(CLetIn) $\mathcal{C}[e_1] \to_l let\ X = e_1\ in\ \mathcal{C}[X]$, $\forall \mathcal{C} \neq []$, if $BV(\mathcal{C}) \cap FV(e_1) = \emptyset$, for $X \in \mathcal{V}$ fresh

This rule introduces non-termination for every program, but it will be useful to reason about the programs, as it leaves the hypersemantics untouched; and because it generalizes the let distribution rules (LetIn$_1$) and (LetIn$_2$). Given $\theta \in CSubst_\perp$:

$$\begin{array}{ll}
[\![(let\ X = e_1\ in\ \mathcal{C}[X])\theta]\!] & \\
= [\![let\ X = e_1\theta\ in\ \mathcal{C}\theta[X]]\!] & \text{variable convention} \\
= \bigcup_{t_1 \in [\![e_1\theta]\!]} [\![(\mathcal{C}\theta[X])[X/t_1]]\!] & \text{lemma 10} \\
= \bigcup_{t_1 \in [\![e_1\theta]\!]} [\![\mathcal{C}\theta[t_1]]\!] & \text{variable convention} \\
= [\![\mathcal{C}\theta[e_1\theta]]\!] & BV(\mathcal{C}) \cap FV(e_1) = \emptyset \\
= [\![(\mathcal{C}[e_1])\theta]\!] & \text{variable convention}
\end{array}$$

But then, as every (LetIn$_1$) step is a particular case of a (CLetIn) step then (LetIn$_1$) leaves the hypersemantics untouched as (CLetIn) does.

(LetIn$_2$) As every (LetIn$_2$) step is a particular case of a (CLetIn) step then (LetIn$_2$) leaves the hypersemantics untouched as (CLetIn) does.
(Contx) By the monotonicity under contexts of the hypersemantics (see [20]).

Now the tools for proving the main results concerning soundness of the simulation are available.

Proof (For Theorem 4). It is straightforward to extend lemma 1 to any number of steps by a simple induction on the length of the derivation. But then $\tau(\mathcal{P}) \vdash e \to^{rt\,*} e'$ implies $[\![e']\!]^{\tau(\mathcal{P})}_{CRWL_{let}} \Subset [\![e]\!]^{\tau(\mathcal{P})}_{CRWL_{let}}$, so $[\![e']\!]^{\tau(\mathcal{P})}_{CRWL_{let}} = [\![e']\!]^{\tau(\mathcal{P})}_{CRWL_{let}}\epsilon \subseteq [\![e]\!]^{\tau(\mathcal{P})}_{CRWL_{let}}\epsilon = [\![e]\!]^{\tau(\mathcal{P})}_{CRWL_{let}}$. On the other hand *b)* is a consequence of *a)*, as $\forall t \in CTerm_\bot$ we have $t \in [\![t]\!]$, so $t \in [\![t]\!] \subseteq [\![e]\!]$, by *a)*.

Proof (For Theorem 5). Assume $e \to^{rt\,*}_{\tau(\mathcal{P})} t$, then by Theorem 4 that implies $e \to^{ct\,*}_{\tau(\mathcal{P})} t$, in other words, $t \in [\![e]\!]^{\tau(\mathcal{P})}$. But $[\![e]\!]^{\tau(\mathcal{P})} = [\![e]\!]^{\mathcal{P}}$ by Theorem 3, hence $t \in [\![e]\!]^{\mathcal{P}}$. In other words, $e \to^{ct\,*}_{\mathcal{P}} t$.

Regarding completeness of $\to^{rt}$ wrt $\to^{ct}$, we will base on the following technical lemmas:

Lemma 13. *For every $e \in Exp$, $t \in CTerm_\bot$ and $p \in CTerm$ linear:*

a) $|e| \sqsubseteq e$.
b) If $t \sqsubseteq |e|$ *then* $t \sqsubseteq e$.
c) Given $\theta \in CSubst_\bot$ *such that* $dom(\theta) \subseteq FV(p)$, *if* $p\theta \sqsubseteq |e|$ *then* $\exists \sigma \in Subst$ *such that* $dom(\sigma) = dom(\theta)$, $p\sigma \equiv e$ *and* $\theta \sqsubseteq \sigma$.

Proof (For lemma 13).

a) By induction on the structure of e:
Base cases
- $e \equiv X$: Then $|e| \equiv X \sqsubseteq X \equiv e$.
- $e \equiv c \in CS^0$: Then $|e| \equiv c \sqsubseteq c \equiv e$.
- $e \equiv f \in FS^0$: Then $|e| \equiv \bot \sqsubseteq e$.

Inductive steps
- $e \equiv c(e_1, \ldots, e_n)$ for $c \in CS$: Then $|e| \equiv c(|e_1|, \ldots, |e_n|) \sqsubseteq_{IH} c(e_1, \ldots, e_n) \equiv e$.
- $e \equiv f(e_1, \ldots, e_n)$ for $f \in FS$: Then $|e| \equiv \bot \sqsubseteq e$.

b) Then $t \sqsubseteq |e| \sqsubseteq e$, by *a)*.
c) By induction on the structure of $p\theta$:
Base cases
- $p\theta \equiv Y \in \mathcal{V}$: Then $p\theta \equiv Y \sqsubseteq |e|$ implies $Y \equiv |e|$ and so $Y \equiv e$. But then we can take $\sigma = \theta$ to get $p\sigma \equiv p\theta \equiv Y \equiv e$, $\theta \sqsubseteq \sigma$ as $\sigma \sqsubseteq \sigma$, and $dom(\sigma) = dom(\theta)$.
- $p\theta \equiv c \in CS^0$: Then $p\theta \equiv c \sqsubseteq |e|$ implies $c \equiv |e|$ and so $c \equiv e$. But then we can take $\sigma = \theta$ to get $p\sigma \equiv p\theta \equiv c \equiv e$, $\theta \sqsubseteq \sigma$ as $\sigma \sqsubseteq \sigma$, and $dom(\sigma) = dom(\theta)$.
- $p\theta \equiv \bot$: Then $p \equiv X \in \mathcal{V}$, and $\theta = [X/\ \bot]$, as $dom(\theta) \subseteq FV(p) = \{X\}$. But then we can choose $\sigma = [X/e]$ to get $p\sigma \equiv X[X/e] \equiv e$, $\theta = [X/\ \bot] \sqsubseteq [X/e] \equiv \sigma$, and $dom(\sigma) = \{X\} = dom(\theta)$.

Inductive steps

- $p\theta \equiv c(s_1, \ldots, s_n)$ with $p \equiv X \in \mathcal{V}$: Then $dom(\theta) \subseteq FV(p)$ implies $dom(\theta) = \{X\}$, so $\theta = [X/X\theta] = [X/p\theta]$. As $\theta \in CSubst_\perp, p \in CTerm$ then $p\theta \in CTerm_\perp$ and so $p\theta \sqsubseteq |e|$ implies $p\theta \sqsubseteq e$ by *b)*. But then we can choose $\sigma = [X/e]$ to get $p\sigma \equiv X[X/e] \equiv e$, $\theta = [X/p\theta] \sqsubseteq [X/e] \equiv \sigma$, and $dom(\sigma) = \{X\} = dom(\theta)$.
- $p\theta \equiv c(p_1\theta, \ldots, p_n\theta)$ with $p \equiv c(p_1, \ldots, p_n)$. Then $p\theta \equiv c(p_1\theta, \ldots, p_n\theta) \sqsubseteq |e|$ implies $|e| \equiv c(|e_1|, \ldots, |e_n|)$ for $e \equiv c(e_1, \ldots, e_n)$ such that $\forall i,\ p_i\theta \sqsubseteq |e_i|$. As p is linear and $dom(\theta) \subseteq FV(p)$ then if for every i we define $\theta_i = \theta|_{FV(p_i)}$ then $\theta = \theta_1 \uplus \ldots \uplus \theta_n$. But then for every i we have $p_i\theta_i \sqsubseteq |e_i|$ to which we can apply the IH to get $\exists \sigma_i \in Subst$ such that $p_i\sigma_i \equiv e_i$, $\theta_i \sqsubseteq \sigma_i$ and $dom(\sigma_i) = dom(\theta_i)$. But as p is linear then $\sigma = \sigma_1 \uplus \ldots \uplus \sigma_n$ is correctly defined and $p\sigma \equiv c(p_1\sigma_1, \ldots, p_n\sigma_n) \equiv c(e_1, \ldots, e_n) \equiv e$, $\theta \sqsubseteq \sigma$ and $dom(\sigma) = dom(\sigma_1) \cup \ldots \cup dom(\sigma_n) = dom(\theta_1) \cup \ldots \cup dom(\theta_n) = dom(\theta)$.

Note that lemma 13 *a)* is not true in general for $e \in LExp$ as $c(\perp) \equiv |let\ X = loop\ in\ c(X)| \not\sqsubseteq let\ X = loop\ in\ c(X)$, so it happens for lemma 13 *b)*, as a consequence. Again lemma 13 is not true in general for $e \in LExp$, just taking $p \equiv c(X), \theta = [X/\perp], e \equiv let\ X = loop\ in\ c(X)$: $p\theta \equiv c(\perp) \sqsubseteq c(\perp) \equiv |let\ X = loop\ in\ c(X)|$, but $\nexists \sigma \in LSusbt$ such that $p\sigma \equiv e$.

The following lemma shows that using the rules for $\rightarrow^{rt}$ except **(Fapp)**, any expression $e \in LExp$ can be transformed to a 'flat' fully developed form with respect to *let*s.

Lemma 14 (Peeling lemma for $\rightarrow^{rt}$). *For every $e \in LExp$ one has*

$$e \rightarrow^{rt\,*} let\ \overline{X = a}\ in\ b$$

such that:

- $\forall a_i \in \overline{a}, a_i \in Exp$ *(i.e., there are no nested lets)*
- $b \in Exp$ *(i.e., it is a let-free body)*
- $\forall a_i \in \overline{a}$, a_i *is function rooted or* $a_i \in FV(let\ \overline{X = a}\ in\ b)$ *(i.e., no applicable binding remains)*

Besides (Fapp) was not used in that derivation (and therefore the $CRWL_{let}$-hypersemantics and the shell remain untouched).

Proof (For lemma 14). Through this proof we will assume α-conversion when needed to fulfil the conditions of application of rules of $\rightarrow^{rt}$. We proceed by induction on the structure of e:

Base cases If $e \equiv Y \in \mathcal{V}$ or $e \equiv h \in \Sigma^0$ then the lemma holds for $e \rightarrow^{rt\,0} e$ with $\overline{X} = \emptyset$.

Inductive steps

• $e \equiv h(e_1, \ldots, e_n)$: Then by IH over each e_i we have $e_i \rightarrow^{rt\,*} let\ \overline{X_i = a_i}\ in\ b_i$ under the conditions stipulated, so we can do:

$$
\begin{array}{ll}
& h(e_1,\ldots,e_n) \\
& \to^{rt\,*} h(let\ \overline{X_1 = a_1}\ in\ b_1,\ldots,let\ \overline{X_n = a_n}\ in\ b_n) \quad (1) \\
& \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ \ldots\ let\ \overline{X_n = a_n}\ in\ h(b_1,\ldots,b_n) \quad (2)
\end{array}
$$

(1) by IH; (2) by (Flat_1^*). But then:

- $\forall a \in \overline{a_1} \cup \ldots \cup \overline{a_n},\ a \in Exp$ by IH.
- $b_1,\ldots,b_n \in Exp$ by IH, so $h(b_1,\ldots,b_n) \in Exp$.
- $\forall a \in \overline{a_1} \cup \ldots \cup \overline{a_n}$, by IH we have that a is function rooted or $a \in FV(let\ \overline{X_i = a_i}\ in\ b_i)$, for the corresponding i. In the latter case that implies we have $a \in FV(h(let\ \overline{X_1 = a_1}\ in\ b_1,\ldots,let\ \overline{X_n = a_n}\ in\ b_n))$ by definition, hence $a \in FV(let\ \overline{X_1 = a_1}\ in\ \ldots\ let\ \overline{X_n = a_n}\ in\ h(b_1,\ldots,b_n))$ as free variables are preserved by (Flat_1) steps, even when put in non trivial contexts (easy to check).

• $e \equiv let\ X = e_1\ in\ e_2$: Then by IH over e_1 and e_2 we can do:

$$
\begin{array}{ll}
& let\ X = e_1\ in\ e_2 \to^{rt\,*} let\ X \\
& = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ (let\ \overline{X_2 = a_2}\ in\ b_2) \quad (1) \\
& \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \quad (2)
\end{array}
$$

(1) by IH; (2) by (Flat_2^*). Then as $b_1 \in Exp$ by IH, we have the following possibilities:

a) b_1 is function rooted or $b_1 \in FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$: Then $b_1 \in Exp$ by IH, and it is easy to check that the other conditions of the lemma are also fulfilled by IH, as (LetIn_1) also preserves free variables.

b) b_1 is constructor rooted or $b_1 \notin FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$. In other words, b_1 is constructor rooted or $b_1 \in BV(let\ \overline{X_1 = a_1}\ in\ let\ X = []\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$. Then we have the following possibilities:

   i) $b_1 \in CTerm$ such that every variable in $var(b_1)$ is bound in its context : Then we can perform a (RBind) step:

$$
\begin{array}{l}
let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2 \\
\to^{rt} let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_2 = a_2[X/b_1]}\ in\ b_2[X/b_1]
\end{array}
$$

   But then:

   - $\forall a \in \overline{a_1},\ a \in Exp$ by IH. Besides $\forall a \in \overline{a_2},\ a \in Exp$ by IH, hence $a[X/b_1] \in Exp$ as $[X/b_1] \in CSubst$.
   - $b_2 \in Exp$ by IH, so $b_2[X/b_1] \in Exp$, as $[X/b_1] \in CSubst$.
   - $\forall a \in \overline{a_1}$, by IH we have that a is function rooted or $a \in FV(let\ \overline{X_1 = a_1}\ in\ b_1)$. In the latter case that implies $a \in FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ let\ \overline{X_2 = a_2}\ in\ b_2)$ by definition, hence $a \in FV(let\ \overline{X_1 = a_1}\ in\ let\ \overline{X_2 = a_2[X/b_1]}\ in\ b_2[X/b_1])$ as free variables are preserved by (RBind) steps, even when put in non trivial contexts (easy to check).
     Besides $\forall a \in \overline{a_2}$, by IH we have that a is function rooted or $a \in FV(let\ \overline{X_2 = a_2}\ in\ b_2)$. In the first case $a_2[X/b_1]$ obviously remains function rooted, and in the latter $a_2[X/b_1] \equiv a_2$, as a_2 was a free in its context, and so it is also free in the new context established by (RBind), which also preserves free variables.

ii) $b_1 \equiv \mathcal{C}[s_1, \ldots, s_n]$ for $\mathcal{C} \neq []$ (otherwise we would be in case *a)*) a many hole c-context and $s_1, \ldots, s_n \in Exp$ the maximal (in the order of positions) subexpressions of b_1 which are function rooted or variables free in their contexts. Those free s_i are also not bound in their contexts, and so we can perform several (LetIn_1) or (LetIn_2) steps:

$$\begin{array}{l} \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ X = b_1\ \mathit{in}\ \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ b_2 \\ \equiv \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ X = \mathcal{C}[s_1, \ldots, s_n]\ \mathit{in}\ \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ b_2 \\ \to^{rt\,*} \mathit{let}\ \overline{X_1 = a_1}\ \mathit{in}\ \mathit{let}\ \overline{Y = s}\ \mathit{in}\ \mathit{let}\ X = \mathcal{C}[\overline{Y}]\ \mathit{in}\ \mathit{let}\ \overline{X_2 = a_2}\ \mathit{in}\ b_2 \end{array}$$

But then we are in the previous case with $\mathcal{C}[\overline{Y}]$ playing the role of b_1, because every $s_i \in Exp$ and besides is function rooted or a free variable, and besides (LetIn_1) and (LetIn_2) preserve free variables.

The following pair of technical but interesting lemmas will be needed to cope with the renaming implicitly introduced by (LetIn_2).

Lemma 15. *For every $p \in CTerm$ linear, $\sigma \in LSubst_\perp$, $\overline{X}, \overline{Y}$ linear and finite tuples of variables and $e \in Exp$ such that $dom(\sigma) \subseteq FV(p)$ if $p\sigma \equiv e[\overline{X/Y}]$ then $\exists \sigma' \in LSubst_\perp$ such that $dom(\sigma') = dom(\sigma)$, $p\sigma' \equiv e$ and $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$. Besides σ' is in the same subset of $LSubst_\perp$ as σ (is total when σ is, is constructed when σ is, ...).*

Proof (For lemma 15). Note that $dom(\sigma') = dom(\sigma) \subseteq FV(p)$ and $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$ does not imply $\sigma'[\overline{X/Y}] = \sigma$, as in general $dom(\sigma'[\overline{X/Y}]) \neq dom(\sigma')$. We proceed by induction on the structure of p:

Base cases

- $p \equiv U \in \mathcal{V}$: Then by hypothesis $U\sigma \equiv p\sigma \equiv e[\overline{X/Y}]$ and $dom(\sigma) \subseteq FV(p) = \{U\}$, hence $\sigma = [U/e[\overline{X/Y}]]$. But then we can take $\sigma' = [U/e]$, with which $p\sigma' \equiv e$, $dom(\sigma') = \{U\} = dom(\sigma)$, and given $Z \in FV(p) = \{U\}$ then $Z = U$ and so $Z\sigma'[\overline{X/Y}] \equiv U[U/e][\overline{X/Y}] \equiv U\sigma \equiv Z\sigma$.
- $p \equiv c \in CS^0$: Then by hypothesis $dom(\sigma) \subseteq FV(p) = \emptyset$, hence $\sigma = \epsilon$. Besides by hypothesis $c \equiv c\epsilon \equiv p\sigma \equiv e[\overline{X/Y}]$, therefore $e \equiv c$. But then we can take $\sigma' = \epsilon$, with which $p\sigma' \equiv c \equiv e$, $dom(\sigma') = \emptyset = dom(\sigma)$, and $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$ trivially as $FV(p) = \emptyset$.

Inductive step Then $p \equiv c(p_1, \ldots, p_n)$: As by hypothesis $p\sigma \equiv e[\overline{X/Y}]$ then it must happen that $e \equiv c(e_1, \ldots, e_n)$ such that $p\sigma \equiv c(p_1\sigma, \ldots, p_n\sigma) \equiv c(e_1[\overline{X/Y}], \ldots, e_n[\overline{X/Y}]) \equiv e[\overline{X/Y}]$. As p is linear then if for each $i \in \{1, \ldots, n\}$ we define $\sigma_i = \sigma|_{FV(p_i)}$ then $\sigma_1 \uplus \ldots \uplus \sigma_n$ is correctly defined and besides $\sigma = \sigma_1 \uplus \ldots \uplus \sigma_n$, as $dom(\sigma) \subseteq FV(p)$ by hypothesis, and $p_i\sigma_i \equiv e_i[\overline{X/Y}]$ for each i. But then we can apply the IH to each i to get some $\sigma'_i \in Subst$ such that $p_i\sigma' \equiv e_i$, $dom(\sigma'_i) = dom(\sigma_i) \subseteq FV(p_i)$ and $\sigma'_i[\overline{X/Y}] = \sigma_i\ [FV(p_i)]$. So $\sigma' = \sigma'_1 \uplus \ldots \uplus \sigma'_n$ is correctly defined and besides:

- $p\sigma' \equiv c(p_1\sigma', \ldots, p_n\sigma') \equiv c(p_1\sigma'_1, \ldots, p_n\sigma'_n) \equiv c(e_1[\overline{X/Y}], \ldots, e_n[\overline{X/Y}]) \equiv e[\overline{X/Y}]$.
- $dom(\sigma') = dom(\sigma'_1) \uplus \ldots \uplus dom(\sigma'_n) = dom(\sigma_1) \uplus \ldots \uplus dom(\sigma_n) = dom(\sigma)$.
- $\sigma'[\overline{X/Y}] = \sigma\ [FV(p)]$ because given $U \in FV(p)$ then, as p is linear, $\exists!\ i \in \{1, \ldots, n\}$ such that $U \in FV(p_i)$. But then $U\sigma'[\overline{X/Y}] \equiv U\sigma'_i[\overline{X/Y}] \equiv U\sigma_i \equiv U\sigma$, as $\sigma'_i[\overline{X/Y}] = \sigma_i\ [FV(p_i)]$ by IH.

Lemma 16. *For any program $\mathcal{P}$, $e \in LExp$, $t \in CTerm_\bot$, $\overline{X}, \overline{Y}$ linear and finite tuples of variables, if $\overline{X} \cap var(t) = \emptyset$ and $\mathcal{P} \vdash e[\overline{X/Y}] \twoheadrightarrow t$ then $\exists t' \in CTerm_\bot$ such that $\mathcal{P} \vdash e \twoheadrightarrow t'$ with a derivation of the same size and structure that $t'[\overline{X/Y}] \equiv t$.*

Proof (For lemma 16). We will prove this lemma for $\overline{X} = \{X\}$ and $\overline{Y} = \{Y\}$, that is "For every $e \in LExp$, $t \in CTerm_\bot$, $X, Y \in \mathcal{V}$ and under any program, if $X \notin var(t)$ and $\mathcal{P} \vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow t$ then $\exists t' \in CTerm_\bot$ such that $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t'$ with a derivation of the same size and structure and $t'[X/Y] \equiv t$.". The extension to finite sets of variables is a trivial induction on the cardinal of those sets. We proceed by induction on the structure of the derivation for $\vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow t$:

Base cases

B If $\vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow \bot$ then $\vdash_{CRWL_{let}} e \twoheadrightarrow \bot$ by **B** and $t'[X/Y] \equiv \bot$ $[X/Y] \equiv \bot \equiv t$.

RR Then $e[X/Y] \in \mathcal{V}$ and we have the following possibilities:

- $e \equiv X$: Then $\vdash_{CRWL_{let}} e[X/Y] \equiv Y \twoheadrightarrow Y \equiv t$, but then $\vdash_{CRWL_{let}} e \equiv X \twoheadrightarrow X \equiv t'$ by **RR** and $t'[X/Y] \equiv X[X/Y] \equiv Y \equiv t$.
- $e \equiv Z \in \mathcal{V}$ such that $Z \neq X$: Then $\vdash_{CRWL_{let}} e[X/Y] \equiv Z \twoheadrightarrow Z \equiv t$, but then $\vdash_{CRWL_{let}} e \equiv Z \twoheadrightarrow Z \equiv t'$ by **RR** and $t'[X/Y] \equiv Z[X/Y] \equiv Z \equiv t$.

Inductive steps

DC Then it must happen $e \equiv c(e_1, \ldots, e_n)$ and the derivation is:

$$\frac{e_1[X/Y] \twoheadrightarrow t_1 \ \ldots \ e_n[X/Y] \twoheadrightarrow t_n}{e[X/Y] \equiv c(e_1[X/Y], \ldots, e_n[X/Y]) \twoheadrightarrow c(t_1, \ldots, t_n) \equiv t} \ DC$$

By IH, for each $i \in \{1, \ldots, n\}$ there must exists some $t'_i \in CTerm_\bot$ such that $\vdash_{CRWL_{let}} e_i \twoheadrightarrow t'_i$ and $t'_i[X/Y] \equiv t_i$. But then we can do:

$$\frac{e_1 \twoheadrightarrow t'_1 \ \ldots \ e_n \twoheadrightarrow t'_n}{e \equiv c(e_1, \ldots, e_n) \twoheadrightarrow c(t'_1, \ldots, t'_n) \equiv t'} \ DC$$

But then $t'[X/Y] \equiv c(t'_1[X/Y], \ldots, t'_n[X/Y]) \equiv c(t_1, \ldots, t_n) \equiv t$.

OR Then it must happen $e \equiv f(e_1, \ldots, e_n)$ such that for some $R = (f(\overline{p}) \to r) \in \mathcal{P}$ and $\theta \in CSusbt_\bot$ we have a derivation like:

$$\frac{e_1[X/Y] \twoheadrightarrow p_1\theta \ \ldots \ e_n[X/Y] \twoheadrightarrow p_n\theta \quad r\theta \twoheadrightarrow t}{e[X/Y] \equiv f(e_1[X/Y], \ldots, e_n[X/Y]) \twoheadrightarrow t} \ OR$$

We assume that R is fresh and $dom(\theta) \subseteq FV(R)$ without loss of generality. But then, as p is linear we can decompose θ as $\theta = \theta_1 \uplus \ldots \theta_n \uplus \theta_{vE}$, where $\theta_i = \theta|_{FV(p_i)}$ and $\theta_{vE} = \theta|_{vExtra(R)}$. Besides, by IH, for each $i \in \{1, \ldots, n\}$ there must exists some $s_i \in CTerm_\bot$ such that $\vdash_{CRWL_{let}} e_i \twoheadrightarrow s_i$ and $s_i[X/Y] \equiv p_i\theta \equiv p_i\theta_i$. Then we can apply lemma 15 to each i to get some $\theta'_i \in CSubst_\bot$ such that $p_i\theta'_i \equiv s_i$, $\theta'_i[X/Y] = \theta_i$ $[var(p_i)]$ and $dom(\theta'_i) = dom(\theta_i) \subseteq FV(p_i)$. But then

$$\begin{aligned} r\theta &\equiv r(\theta_1 \uplus \ldots \uplus \theta_n \uplus \theta_{vE}) \\ &\equiv r((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \\ &\qquad \uplus \theta_{vE}) \end{aligned}$$

Furthermore by lemma 4 we assume that the set $\mathcal{U}$ of the free variables used in $\vdash_{CRWL_{let}} e[X/Y] \twoheadrightarrow t$ fulfils $\mathcal{U} \subseteq FV(e[X/Y]) \cup FV(t)$. As obviously $X \notin FV(e[X/Y])$, and $X \notin FV(t)$ by hypothesis, then $X \notin \mathcal{U}$. Besides $FV(r\theta) = FV(r(\theta_1 \uplus \ldots \uplus \theta_n \uplus \theta_{vE})) \subseteq \mathcal{U}$, as $r\theta$ is used in the derivation for $e[X/Y] \twoheadrightarrow t$, and $X \notin FV(r)$ as r is part of the fresh instance, hence $X \notin vran(\theta_{vE})$. With this we will prove that $r((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \uplus \theta_{vE}) \equiv r(\theta'_1 \uplus \ldots \uplus \theta'_n \uplus \theta_{vE})[X/Y]$. Given $Z \in FV(r) \subseteq var(p_1) \uplus \ldots \uplus var(p_n) \uplus vExtra(R)$, we have the following possibilities:

- If $Z \in var(p_i)$ for some $i \in \{1, \ldots, n\}$ then

$$\begin{array}{l} Z((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \\ \qquad\qquad\qquad\qquad\qquad\qquad \uplus \theta_{vE}) \\ \equiv Z(\theta'_i[X/Y])|_{var(p_i)} \equiv (Z\theta'_i)[X/Y] \\ \equiv Z(\theta'_1 \uplus \ldots \uplus \theta'_n \uplus \theta_{vE})[X/Y] \end{array}$$

- If $Z \in vExtra(R)$ then $Z((\theta'_1[X/Y])|_{var(p_1)} \uplus \ldots \uplus (\theta'_n[X/Y])|_{var(p_n)} \uplus \theta_{vE}) \equiv Z\theta_{vE}$. But $Z \in FV(r)$ which is part of the fresh instance, so $Z \neq X$, and $X \notin vran(\theta_{vE})$ as we saw above, hence $Z\theta_{vE} \equiv (Z\theta_{vE})[X/Y] \equiv Z(\theta_{vE} \uplus \ldots \theta'_n \uplus \theta_{vE})[X/Y]$.

So $\vdash_{CRWL_{let}} r(\theta'_1 \uplus \ldots \uplus \theta'_n \uplus \theta_{vE})[X/Y] \equiv r\theta \twoheadrightarrow t$ and we can apply the IH to get $\vdash_{CRWL_{let}} r(\theta'_1 \uplus \ldots \theta'_n \uplus \theta_{vE}) \twoheadrightarrow t'$ under the conditions stipulated. But then we can do:

$$\frac{e_i \twoheadrightarrow s_i \equiv p_i\theta'_i \quad r(\theta'_1 \uplus \ldots \theta'_n \uplus \theta_{vE}) \twoheadrightarrow t'}{e \equiv f(e_1, \ldots, e_n) \twoheadrightarrow t'}\ OR$$

for $i = 1..n$

Let Then it must happen $e \equiv let\ Z = e_1\ in\ e_2$ and the derivation is:

$$\frac{e_1[X/Y] \twoheadrightarrow t_1 \quad e_2[X/Y][Z/t_1] \twoheadrightarrow t}{e[X/Y] \equiv let\ Z = e_1[X/Y]\ in\ e_2[X/Y] \twoheadrightarrow t}\ Let$$

Then we can apply the IH over $\vdash_{CRWL_{let}} e_1[X/Y] \twoheadrightarrow t_1$ to get $\vdash_{CRWL_{let}} e_1 \twoheadrightarrow t'_1$ such that $t'_1[X/Y] \equiv t_1$, under the conditions stipulated. Furthermore by variable convention $Z \notin dom([X/Y]) \cup vran([X/Y])$, and so we can apply the substitution lemma to get $e_2[X/Y][Z/t_1] \equiv e_2[X/Y][Z/t'_1[X/Y]] \equiv e_2[Z/t'_1][X/Y]$. But then we have $e_2[Z/t'_1][X/Y] \equiv e_2[X/Y]\ [Z/t_1] \twoheadrightarrow t$ and we can apply the IH to get $\vdash_{CRWL_{let}} e_2[Z/t'_1] \twoheadrightarrow t'$ under the conditions stipulated, and we can do:

$$\frac{e_1 \twoheadrightarrow t'_1 \quad e_2[Z/t'_1] \twoheadrightarrow t'}{e \equiv let\ Z = e_1\ in\ e_2 \twoheadrightarrow t'}\ Let$$

Finally we are ready to prove lemma 2 with the help of the auxiliary lemmas above:

Proof (For lemma 2). Through this proof we will assume α-conversion when needed to fulfil the conditions of application of rules of $\rightarrow^{rt}$. We proceed by induction on the size of the $CRWL_{let}$ derivation for $\mathcal{P} \vdash_{CRWL_{let}} e \twoheadrightarrow t$, measured as the number of rules of $CRWL_{let}$ applied. Let us see which rule was applied at the root of that derivation:

Base cases This cases correspond to $e \twoheadrightarrow \bot$ by **B**, $X \twoheadrightarrow X$ by **RR**, for $X \in \mathcal{V}$, and $c \twoheadrightarrow c$ by **DC**, for $c \in CS^0$. In any of these cases $e \rightarrow^{rt\,0} e \equiv e'$ fulfils the conditions of the lemma, because then $\bot \sqsubseteq |e|$, $X \sqsubseteq X \equiv |X|$ and $c \sqsubseteq c \equiv |c|$.

Inductive steps

DC Then we have

$$\frac{e_1 \twoheadrightarrow t_1 \quad \ldots \quad e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\ DC$$

Then by IH over each $e_i \twoheadrightarrow t_i$ then $e_i \rightarrow^{rt\,*} e'_i$ for some $e'_i \in LExp$ such that $t_i \sqsubseteq |e'_i|$. But then $c(e_1,\ldots,e_n) \rightarrow^{rt\,*} c(e'_1,\ldots,e'_n)$, so we are done as $\forall i, t_i \sqsubseteq |e'_i|$ implies $c(t_1,\ldots,t_n) \sqsubseteq c(|e'_1|,\ldots,|e'_n|) \equiv |c(e'_1,\ldots,e'_n)|$.

OR For the sake of clarity we will present the proof for the case of an application of $f \in FS^1$, the adaptation of this proof to zero or more than one argument is straightforward. Then we have:

$$\frac{e_1 \twoheadrightarrow p_1\theta \quad r\theta \twoheadrightarrow t}{f(e_1) \twoheadrightarrow t}\ OR$$

for some rule $R = (f(p_1) \to r) \in \mathcal{P}$ and $\theta \in CSusbt_\bot$. By IH over $e_1 \twoheadrightarrow p_1\theta$ then $e_1 \rightarrow^{rt\,*} e'_1$ for some $e'_1 \in LExp$ such that $p_1\theta \sqsubseteq |e'_1|$. But then:

$$\begin{array}{ll} f(e_1) \rightarrow^{rt\,*} f(e'_1) & \text{by IH} \\ \rightarrow^{rt\,*} f(let\ \overline{X_1 = a_1}\ in\ b_1) & \text{by the peeling lemma 14} \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ f(b_1) & \text{by } (Flat^*_1) \end{array}$$

By the conditions of the peeling lemma we can decompose $\overline{a_1}$ as $\overline{a_1} = \overline{a_1^f} \uplus \overline{a_1^v}$, where $\overline{a_1^f}$ contains those $a \in \overline{a_1}$ which are function rooted and $\overline{a_1^v}$ contains those which are free variables (as (Flat_1) preserves the free variables these remain free in $let\ \overline{X_1 = a_1}\ in\ f(b_1)$). But as in the derivation of the peeling lemma (Fapp) was not applied then by lemma 2 the shell was preserved and so

$$\begin{array}{l} p_1(\theta|_{FV(p_1)}) \equiv p_1\theta \sqsubseteq |e'_1| \equiv |let\ \overline{X_1 = a_1}\ in\ b_1| \\ \equiv |b_1|[\overline{X_1^f/\ \bot}, \overline{X_1^v/a_1^v}] \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]| \end{array}$$

But, as $p_1 \in CTerm$ is lineal, $\theta|_{FV(p_1)} \in CSubst_\bot$ with $dom(\theta|_{FV(p_1)}) \subseteq FV(p_1)$, $b_1 \in Exp$ by the peeling lemma and so $b_1[\overline{X_1^v/a_1^v}] \in Exp$, and $p_1(\theta|_{FV(p_1)}) \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]|$, then we can apply lemma 13 to get some $\sigma_1 \in Subst$ such that $dom(\sigma_1) = dom(\theta|_{FV(p_1)}) \subseteq FV(p_1)$, $p_1\sigma_1 \equiv b_1[\overline{X_1^v/a_1^v}]$ and $\theta|_{FV(p_1)} \sqsubseteq \sigma_1$. But then the conditions in lemma 15 are also fulfilled and so we can apply it to get some $\sigma'_1 \in Subst$ such that $dom(\sigma'_1) = dom(\sigma_1) \subseteq FV(p_1)$, $p_1\sigma'_1 \equiv b_1$ and $\sigma'_1[\overline{X_1^v/a_1^v}] = \sigma_1\ [FV(p_1)]$.

Without loss of generality we assume $dom(\theta) \subseteq FV(f(p_1) \to r)$, so $\theta = \theta|_{FV(p_1)} \uplus \theta|_{vExtra(R)}$. Now we can define $\theta' \in CSubst$ such that $\theta|_{vExtra(R)} \sqsubseteq \theta'$ and $dom(\theta') = vExtra(R)$, just replacing every $\bot$ introduced by $\theta|_{vExtra(R)}$ in its range with some constant or fresh variable. But then $\sigma_1 \uplus \theta'$ is correctly defined and besides $\theta \sqsubseteq \sigma_1 \uplus \theta'$, hence $r\theta \twoheadrightarrow t$ implies $r(\sigma_1 \uplus \theta') \twoheadrightarrow t$ with a derivation of the same size or smaller. Besides:

$$\begin{array}{ll} r(\sigma_1 \uplus \theta') \equiv r((\sigma_1)|_{FV(p_1)} \uplus \theta') & (1) \\ \equiv r((\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \uplus \theta') & (2) \\ \equiv r(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}] & (3) \end{array}$$

(1) as $dom(\sigma_1) \subseteq FV(p_1)$;
(2) as $\sigma'_1[\overline{X_1^v/a_1^v}] = \sigma_1\ [FV(p_1)]$;
(3) Because given $Z \in FV(r) \subseteq FV(p_1) \uplus vExtra(R)$:
- If $Z \in FV(p_1)$ then $Z((\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \uplus \theta') \equiv Z(\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \equiv (Z\sigma'_1)[\overline{X_1^v/a_1^v}] \equiv Z(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}]$.
- Otherwise $Z \in vExtra(R)$,
  but then $Z((\sigma'_1[\overline{X_1^v/a_1^v}])|_{FV(p_1)} \uplus \theta') \equiv Z\theta'$. But without loss of generality we assume that R is a fresh instance and so Z is fresh as it is part of R and $Z \notin \overline{X_1^v}$; besides $\overline{X_1^v} \cap vran(\theta') = \emptyset$ by lemma 5, as the variables in $\overline{X_1^v}$ either are bound in a subderivation of $\vdash_{CRWL_{let}} f(e_1) \twoheadrightarrow t$ or are fresh and introduced by $\rightarrow^{rt}$. Hence $Z\theta' \equiv Z\theta'[\overline{X_1^v/a_1^v}] \equiv Z(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}]$.

So $\vdash_{CRWL_{let}} r(\sigma'_1 \uplus \theta')[\overline{X_1^v/a_1^v}] \equiv r(\sigma_1 \uplus \theta') \twoheadrightarrow t$ and $\overline{X_1^v} \cap var(t) = \emptyset$ by lemma 5, as the variables in $\overline{X_1^v}$ either are bound in a subderivation of $\vdash_{CRWL_{let}} f(e_1) \twoheadrightarrow t$ or are fresh and introduced by $\rightarrow^{rt}$. Then we can apply lemma 16 to get $\vdash_{CRWL_{let}} r(\sigma'_1 \uplus \theta') \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\perp$ such that $t'[\overline{X_1^v/a_1^v}] \equiv t$. Finally we can apply the IH to this derivation to get $r(\sigma'_1 \uplus \theta') \rightarrow^{rt\,*} e'$ for some $e' \in LExp$ such that $t' \sqsubseteq |e'|$, so we can do:

$$\begin{array}{ll} let\ \overline{X_1 = a_1}\ in\ f(b_1) \equiv let\ \overline{X_1 = a_1}\ in\ f(p_1\sigma'_1) & \\ \equiv let\ \overline{X_1 = a_1}\ in\ f(p_1(\sigma'_1 \uplus \theta')) \rightarrow^{rt} let\ \overline{X_1 = a_1}\ in\ r(\sigma'_1 \uplus \theta') & \text{by (Fapp)} \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ e' & \text{by IH} \end{array}$$

Now, as $\overline{a_1^v} \subseteq \mathcal{V}$ then $t' \sqsubseteq |e'|$ implies $t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ and so $t \equiv t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$. Besides $\overline{X_1^f} \cap var(t) = \emptyset$ for the same reasons that $\overline{X_1^v} \cap var(t) = \emptyset$, but then $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies that any occurrence in $|e'|[\overline{X_1^v/a_1^v}]$ of some $X \in \overline{X_1^f}$ corresponds to an occurrence of $\perp$ in t, in the same position. Hence $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\perp}] \equiv |let\ \overline{X_1 = a_1}\ in\ e'|$.

Let Then we have

$$\frac{e_1 \twoheadrightarrow t_1 \quad e_2[X/t_1] \twoheadrightarrow t}{let\ X = e_1\ in\ e_2 \twoheadrightarrow t}\ Let$$

Then by IH over $e_1 \twoheadrightarrow t_1$ then $e_1 \rightarrow^{rt\,*} e'_1$ for some $e'_1 \in LExp$ such that $t_1 \sqsubseteq |e'_1|$, so we can do:

$$\begin{array}{l} let\ X = e_1\ in\ e_2 \rightarrow^{rt\,*} let\ X = e'_1\ in\ e_2 \\ \rightarrow^{rt\,*} let\ X = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ e_2 \quad (1) \\ \rightarrow^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \quad (2) \end{array}$$

(1) by the peeling lemma 14; (2) by (Flat_2^*). By the conditions of the peeling lemma we can decompose $\overline{a_1}$ as $\overline{a_1} = \overline{a_1^f} \uplus \overline{a_1^v}$, where $\overline{a_1^f}$ contains those $a \in \overline{a_1}$ which are function rooted and $\overline{a_1^v}$ contains those which are free variables (as (Flat_2) preserves the free variables these remain free in $let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2$). But as in the derivation of the peeling lemma (Fapp) was not applied then by lemma 2 the shell was preserved and so

$$t_1 \sqsubseteq |e'_1| \equiv |let\ \overline{X_1 = a_1}\ in\ b_1| \equiv |b_1|\overline{[X_1^f/\perp}, \overline{X_1^v/a_1^v}] \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]|$$

Now we have several possibilities, taking into account that $b_1 \in Exp$, by the conditions of the peeling lemma:

a) $b_1 \in CTerm$ such that every variable in $var(b_1)$ is bound in its context : Then as $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]|$ we have $t_1 \sqsubseteq b_1[\overline{X_1^v/a_1^v}]$ by lemma 13, so $[X/t_1] \sqsubseteq [X/b_1[\overline{X_1^v/a_1^v}]]$. Hence we have $\vdash_{CRWL_{let}} e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2[X/b_1[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$ with a derivation of the same size or smaller. Besides, $\overline{X_1^v} \cap FV(e_2) = \emptyset$, by the conditions in (Flat$_2$), so $e_2[X/b_1[\overline{X_1^v/a_1^v}]] \equiv e_2[\overline{X_1^v/a_1^v}][X/b_1[\overline{X_1^v/a_1^v}]] \equiv e_2[X/b_1][\overline{X_1^v/a_1^v}]$ by the substitution lemma, as $X \notin (dom([\overline{X_1^v/a_1^v}]) \cup vran([\overline{X_1^v/a_1^v}]))$ by α-conversion. We can do this conversion because $let\ X = (let\ \overline{X_1 = a_1}\ in\ b_1)\ in\ e_2$ was an intermediate expression, in which we have $X \notin FV(\overline{X_1 = a_1}\ in\ b_1)$ because of the abstense of recursive *lets*. So $\vdash_{CRWL_{let}} e_2[X/b_1]\ [\overline{X_1^v/a_1^v}] \equiv e_2[X/b_1[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$, and then we can apply lemma 16 to get $\vdash_{CRWL_{let}} e_2[X/b_1] \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\perp$ such that $t'[\overline{X_1^v/a_1^v}] \equiv t$. Finally we can apply the IH to this derivation to get $e_2[X/b_1] \to^{rt\,*} e'$ for some $e' \in LExp$ such that $t' \sqsubseteq |e'|$, so we can do:

$$\begin{array}{ll} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \to^{rt} let\ \overline{X_1 = a_1}\ in\ e_2[X/b_1] & \text{by (RBind)} \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ e' & \text{by IH} \end{array}$$

Now, as $\overline{a_1^v} \subseteq \mathcal{V}$ then $t' \sqsubseteq |e'|$ implies $t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ and so $t \equiv t'[\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$. Besides we can prove that $\overline{X_1^f} \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies that any occurrence in $|e'|[\overline{X_1^v/a_1^v}]$ of some $X \in \overline{X_1^f}$ corresponds to an occurrence of $\perp$ in t, in the same position. Hence $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}]$ implies $t \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\perp}] \equiv |let\ \overline{X_1 = a_1}\ in\ e'|$.

b) $b_1 \notin CTerm$ or $b_1 \in CTerm$ but some variable in b_1 is not bound in its context.

i) If b_1 is function rooted then $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]| \equiv \perp$, hence $t_1 \equiv \perp$ and $\vdash_{CRWL_{let}} e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2 \twoheadrightarrow t$ with a derivation of the same size or smaller, to which we can apply the IH to get $e_2 \to^{rt\,*} e'$ for some $e' \in LExp$ such that $t \sqsubseteq |e'|$, so we can do:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e' \quad \text{by IH} \end{array}$$

Now we can prove that $(\overline{X_1^f} \cup \overline{X_1^v} \cup \{X\}) \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|$ implies that any occurrence in $|e'|$ of some $Y \in \overline{X_1^f} \cup \overline{X_1^v} \cup \{X\}$ corresponds to an occurrence of $\perp$ in t, in the same position. Hence $t \sqsubseteq |e'|$ implies $t \sqsubseteq |e'|[\overline{X_1^v/\perp}, \overline{X_1^f/\perp}, X/\perp] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\perp}, X/\perp] \equiv |let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ e'|$, as b_1 is function rooted.

ii) If $b_1 \equiv Y \in FV(let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2)$ then $Y \notin \overline{X_1}$ and $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]| \equiv |Y[\overline{X_1^v/a_1^v}]| \equiv |Y| \equiv Y$. So we can apply lemma 16 to get that $\vdash_{CRWL_{let}} e_2[X/Y] \equiv e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2 \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\perp$ such that

$t'[X/Y] \equiv t$, to which we can apply the IH to get $e_2 \to^{rt\,*} e'$ for some $e' \in LExp$ such that $t' \sqsubseteq |e'|$, so we can do:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e' \quad \text{by IH} \end{array}$$

Now $t' \sqsubseteq |e'|$ implies $t'[X/Y] \sqsubseteq |e'|[X/Y]$ and so $t \equiv t'[X/Y] \sqsubseteq |e'|[X/Y]$. Besides we can prove that $(\overline{X_1^f} \cup \overline{X_1^v}) \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|[X/Y]$ implies that any occurrence in $|e'|[X/Y]$ of some $Z \in \overline{X_1^f} \cup \overline{X_1^v}$ corresponds to an occurrence of $\bot$ in t, in the same position. Hence $t \sqsubseteq |e'|[X/Y]$ implies $t \sqsubseteq |e'|[\overline{X_1^v/\bot}, \overline{X_1^f/\bot}, X/Y] \sqsubseteq |e'|[\overline{X_1^v/a_1^v}, \overline{X_1^f/\bot}, X/Y] \equiv |let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ e'|$.

iii) $b_1 \equiv \mathcal{C}[s_1, \ldots, s_n]$ for $\mathcal{C} \neq []$ a many hole c-context and $s_1, \ldots, s_n \in Exp$ the maximal (in the order of positions) subexpressions of b_1 which are function rooted or variables free in their contexts. Those free s_i are also not bound in their contexts, and so we can perform several (LetIn_1) or (LetIn_2) steps:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ X = b_1\ in\ e_2 \\ \equiv let\ \overline{X_1 = a_1}\ in\ let\ X = \mathcal{C}[s_1, \ldots, s_n]\ in\ e_2 \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ let\ X = \mathcal{C}[\overline{Y}]\ in\ e_2 \\ \to^{rt} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e_2[X/\mathcal{C}[\overline{Y}]] \end{array}$$

the first step by $((\text{LetIn}_1 \mid \text{LetIn}_2)^*)$ and the second by (RBind). Now we can decompose $\overline{s}$ as $\overline{s} = \overline{s^f} \uplus \overline{s^v}$, where $\overline{s^f}$ contains those $s_i \in \overline{s}$ which are function rooted and $\overline{s^v}$ contains those which are free variables. Then $t_1 \sqsubseteq |b_1[\overline{X_1^v/a_1^v}]| \equiv |(\mathcal{C}[\overline{s}])[\overline{X_1^v/a_1^v}]| \sqsubseteq |(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]|$, by lemma 3, as $|\overline{s^f}| = \overline{\bot}$ because those are function rooted. But then $t_1 \sqsubseteq (\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]$ by lemma 13, hence $\vdash_{CRWL_{let}} e_2[X/t_1] \twoheadrightarrow t$ implies $\vdash_{CRWL_{let}} e_2[X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$ with a derivation of the same size or smaller.
Besides, $\overline{X_1^v} \cap FV(e_2) = \emptyset$, by the conditions in ($\text{Flat}_2$), so

$$\begin{array}{l} e_2[X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \equiv e_2[\overline{X_1^v/a_1^v}][X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \\ \equiv e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]][\overline{X_1^v/a_1^v}] \end{array}$$

by the substitution lemma, as $X \notin (dom([\overline{X_1^v/a_1^v}]) \cup vran([\overline{X_1^v/a_1^v}]))$ by α-conversion. So we can apply lemma 16 to get that $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]][\overline{X_1^v/a_1^v}] \equiv e_2[X/(\mathcal{C}[\overline{s^v}, \overline{Y^f}])[\overline{X_1^v/a_1^v}]] \twoheadrightarrow t$ implies that $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]] \twoheadrightarrow t'$ with a derivation of the same size, for some $t' \in CTerm_\bot$ such that $t'[\overline{X_1^v/a_1^v}] \equiv t$.
Furthermore, as $\overline{Y}$ are fresh and linear then

$$\begin{array}{l} e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]] \equiv e_2[X/(\mathcal{C}[\overline{Y}])[\overline{Y^v/s^v}]] \\ \equiv e_2[\overline{Y^v/s^v}][X/(\mathcal{C}[\overline{Y}])[\overline{Y^v/s^v}]] \\ \equiv e_2[X/\mathcal{C}[\overline{Y}]][\overline{Y^v/s^v}] \end{array}$$

because $\overline{Y} \cap FV(e_2) = \emptyset$ by the freshness of $\overline{Y}$, and by the substitution lemma, as $X \notin (dom([\overline{Y^v/s^v}]) \cup vran([\overline{Y^v/s^v}]))$. So we can apply lemma 16 to get that $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{Y}]][\overline{Y^v/s^v}] \equiv e_2[X/\mathcal{C}[\overline{s^v}, \overline{Y^f}]] \twoheadrightarrow t'$ implies $\vdash_{CRWL_{let}} e_2[X/\mathcal{C}[\overline{Y}]] \twoheadrightarrow t''$ with a derivation of the same size, for some $t'' \in CTerm_\bot$ such that $t''[\overline{Y^v/s^v}] \equiv t'$. We can apply the IH to that derivation to get:

$$\begin{array}{l} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e_2[X/\mathcal{C}[\overline{Y}]] \\ \to^{rt\,*} let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e' \end{array}$$

Now, as $\overline{a_1^v} \cup \overline{s^v} \subseteq \mathcal{V}$ and $t'' \sqsubseteq |e'|$ by IH then $t''[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ and so

$$t \equiv t'[\overline{X_1^v/a_1^v}] \equiv t''[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}] \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$$

Besides we can prove that $(\overline{X_1^f} \cup \overline{Y^f}) \cap var(t) = \emptyset$ in the same way we did in the **OR** case, but then $t \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ implies that any occurrence in $|e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ of some $Z \in \overline{X_1^f} \cup \overline{Y^f}$ corresponds to an occurrence of $\bot$ in t, in the same position. Hence $t \sqsubseteq |e'|[\overline{Y^v/s^v}][\overline{X_1^v/a_1^v}]$ implies $t \sqsubseteq |e'|[\overline{Y^v/s^v}, \overline{Y^f/\bot}, \overline{X_1^v/a_1^v}, \overline{X_1^f/\bot}] \equiv |let\ \overline{X_1 = a_1}\ in\ let\ \overline{Y = s}\ in\ e'|$.

Proof (For Theorem 6).
a) Let $\mathcal{P}$ be a program, $e \in LExp$, $t \in CTerm$ and assume $\mathcal{P} \vdash e \twoheadrightarrow t$, which implies $\tau(\mathcal{P}) \vdash e \twoheadrightarrow t$. Lemma 2 ensures that $e \to^{rt\,*}_{\tau(\mathcal{P})} e'$ for some $e' \in LExp$ such that $t \sqsubseteq |e'|$. Since t is total, $t = |e'|$, and $|e'|$ does not contain $\bot$ and therefore no function application. Using this fact to interpret the conlusion of the peeling lemma 14 applied to e', we obtain $e' \to^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = a}\ in\ b$ where all a must free variables and b must be a c-term, say t'. But then we have $e \to^{rt\,*}_{\tau(\mathcal{P})} e' \to^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = a}\ in\ t'$. All bindings $Y = a$ corresponding to Y's not occurring in t' can disappear by **(Elim)**, and therefore we obtain $e \to^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = a}\ in\ t'$ where all remaining Y are variables occurring in t'. Since the reductions made by the peeling lemma and the **(Elim)** rule do not change shells, we have $t = |e'| = |let\ \overline{Y = a}\ in\ t'| = t'[\overline{Y/a}]$, as desired. Notice that since all remaining Y occurred in t', all the a occur (free) in $t'[\overline{Y/a}]$, and therefore in t.
b) Notice simply that since t is ground, the set of bindings in the expression $let\ \overline{Y = a}\ in\ t'$ given by *a)* must be empty, and morevover $t' = t$.

Proof (For Theorem 7).
Let $\mathcal{P}$ be a program, $e \in LExp$, $t \in CTerm_\bot$.

a) The left to right implication is Lemma 2. For $\Leftarrow$, assume $e \to^{rt\,*}_{\tau(\mathcal{P})} e'$, for some $|e'| \sqsupseteq t$. By Theorem 4, $[\![e']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$. Now, since $|e'| \in [\![e']\!]^{\tau(\mathcal{P})}$, we have $|e'| \in [\![e]\!]^{\tau(\mathcal{P})}$. As $|e'| \sqsupseteq t$, a basic property of CRWL-semantics ensures that $t \in [\![e]\!]^{\tau(\mathcal{P})}$, which exactly means that $\mathcal{P} \vdash e \twoheadrightarrow t$.

b) Assume t is total. We have the equivalences $\mathcal{P} \vdash e \twoheadrightarrow t \ \Leftrightarrow\ e \to^{ct\,*}_{\mathcal{P}} t$ and $\mathcal{P} \vdash e \twoheadrightarrow t \ \Leftrightarrow\ \tau(\mathcal{P}) \vdash e \twoheadrightarrow t$ by Theor. 7 of [20] and Theor. 3 respectively. It remains to prove that $\tau(\mathcal{P}) \vdash e \twoheadrightarrow t \ \Leftrightarrow\ e \to^{rt\,*}_{\tau(\mathcal{P})} let\ \overline{Y = X}\ in\ t'$ for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$

and $\overline{X} \subseteq FV(t)$. The implication $\Rightarrow$ is part *a)* of Theor. 6. For $\Leftarrow$ we reason as follows: assume $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}$ *let* $\overline{Y = X}$ *in* t' for some $t' \in CTerm$ with $t'[\overline{Y/X}] \equiv t$ and $\overline{X} \subseteq FV(t)$. Since $e \rightarrow^{rt\,*}_{\tau(\mathcal{P})}$ *let* $\overline{Y = X}$ *in* t', part *a)* of Theor. 4 ensures that $[\![let\ \overline{Y = X}\ in\ t']\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$. But it it easy to prove in the $CRWL_{let}$ framework that $[\![let\ \overline{Y = X}\ in\ t']\!]^{\tau(\mathcal{P})} = [\![t'[Y/X]]\!]^{\tau(\mathcal{P})}$. As $t'[\overline{Y/X}] \equiv t$, we have $[\![t]\!]^{\tau(\mathcal{P})} \subseteq [\![e]\!]^{\tau(\mathcal{P})}$. Finally, $t \in [\![t]\!]$ implies $t \in [\![e]\!]^{\tau(\mathcal{P})}$, which precisely means $\tau(\mathcal{P}) \vdash e \twoheadrightarrow t$, as desired.

c) It follows directly from *b)*, taking into account that the set $\overline{X}$ must be empty since t is ground and $\overline{X} \subseteq FV(t) = \emptyset$.

8.1.11 A Fully Abstract Semantics for Constructor Systems (Extended version)

A Fully Abstract Semantics for Constructor Systems *

(Extended version)

Tech. Rep. SIC-2-09, 2009

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. Constructor-based term rewriting systems are a useful subclass of TRS, in particular for programming purposes. In this kind of systems constructors determine a universe of values, which are the expected output of the computations. Then it would be natural to think of a semantics associating each expression to the set of its reachable values. Somehow surprisingly, the resulting semantics has poor properties, for it is not compositional nor fully abstract when non-clonfluent systems are considered. In this paper we propose a novel semantics for expressions in constructor systems, which is compositional and fully abstract (with respect to sensible observation functions, in particular the set of reachable values for an expression), and therefore can serve as appropriate basis for semantic based analysis or manipulation of such kind of rewrite systems.

1 Introduction

Constructor based term rewriting systems (CS) are an important subclass of TRS. The use of CS for programming has been frequently connected to the requirement of confluence. By these days this is not necessarily so, and many proposals (see e.g.[12, 3, 9, 8, 10]) drop the requirement of confluence and/or termination.

On the other hand, it is widely accepted that an adequate semantics constitutes an excellent companion to any programming language. In the case of CS, an 'obvious' notion of semantics comes from defining the denotation of an expression e as the set of values reachable from e by rewriting. The notion of 'values' could be made concrete in different manners: constructor terms, outer constructor part of expressions or normal forms. Two questions arise:
– Is the semantics compositional? In our case: is the semantics of an expression determined by the semantics of its subexpressions?
– Does it capture observational equivalence? That is: for two semantically equivalent expressions e, e', is it ensured that we will *observe* the same behavior when e, e' are put in the same context? This depends on a criterion of what can be observed from an expression. In the constructor discipline point of view, one is mostly interested again in observing which constructor terms (or outer stable constructor part) can be reached by rewriting.

Somehow surprisingly, the answers is negative for the 'obvious' semantics:

Example 1. Consider the constructors a, b, c, d and the non-confluent program

$$f(c(X)) \to d(X, X) \quad amb(X, Y) \to X \quad amb(X, Y) \to Y$$

The expressions $e \equiv c(amb(a, b))$ and $e' \equiv amb(c(a), c(b))$ reach by rewriting exactly the same constructor values, namely $c(a)$ and $c(b)$. However, this does not ensure that e, e' behave the same when put in the same context. For instance, $f(e)$ can be rewritten to the constructor values $d(a, a), d(a, b), d(b, a), d(b, b)$ while $f(e')$ only to $d(a, a)$ and $d(b, b)$. More in general, this works starts by remarking that *knowing the constructor values of an expression e is not enough information to know the constructor values of $\mathcal{C}[e]$ for any given context $\mathcal{C}$.* The same example shows that the remark remains true if we replace 'constructor value' by 'normal form' or 'outer constructor part'. Using standard terminology (see Sect. 4 for definitions) all those semantics are not compositional, sound nor fully abstract.

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), Promesas-CAM (S-0505/TIC/0407) and STAMP (TIN2008-06622-C03-01/TIN).

The aim of our work can be made clear now: to define a semantics for CS that is fully abstract (compositionality and soundness will come along the way) wrt the observability criterion of reachable constructor terms.

Our starting insight is that, to recover compositionality, the semantics must not collect a *flat* set of reachable values, like is $\{c(a), c(b)\}$ for $c(amb(a,b))$, but rather a more structured and 'packaged' representation, where constructors can be applied to sets, as to reflect more appropriately the matching capabilities of expressions. In our example, and disregarding for the moment some technical details, the denotation of $c(amb(a,b))$ will be the singleton 'package' $\{c(\{a,b\})\}$, reflecting the fact that $c(amb(a,b))$ can match $c(X)$ without reducing $amb(a,b)$, while the denotation of $amb(c(a), c(b))$ will be the two-element package $\{c(a), c(b)\}$. Technically, things will be a bit more complicated (see Sect. 3), in particular due to the possibility of non-termination, that yields possibly infinite sets, and will require expressing some kind of *partial* values in the semantics.

Related work Not too much attention has been paid to the issue of semantics of TRS, at least when compared to the huge amount of research in the fields of TRS and of semantics of programming languages in general. There are nevertheless some works to be mentioned.

In [6], Boudol develops a deep theory of the space of computations of left-linear TRS and provides a computational semantics based on continuous algebras. However, his semantics still associates an expression with a flat set of (possibly infinite) values, thus presenting the problems of our Ex. 1. Moreover, [5, 15] demonstrate that there are problems with achieving full abstraction for non-terminating non-deterministic systems, if the semantics is based on fixpoints and infinite (limit) values (our semantics will avoid them). In [2] a compositional semantics for conditional TRS is presented. Compositionality is understood in a different sense, related to the issue of joining programs. In addition, the considered programs are canonical (confluent and terminating). In [1], an abstract diagnosis scheme for functional programs modeled as TRS is developed, based on some notions of semantics that again collect results of individual computations. The semantics characterization of narrowing given in [11] includes a semantics for TRS, but most of the interesting results are for confluent ones. On the other hand, the cited papers give a more general treatment of variables, which have a passive role in our paper, behaving almost as constants.

With respect to the nesting of sets inside constructor symbols, a similar idea appears in [4], to improve the efficiency of functional logic computations, in [7] as part of the design of a functional programming implementation of functional logic languages, and in [13] as a mean for improving the programming of non-determinism in a Haskell-like ambient. All these works are much more oriented to practice, far from the aims and results of our present work. Moreover, the setting is not the same: functional logic programming for the two first (with a *call-time choice* semantics [9], having essential differences with standard rewriting) and functional programming for the last one.

The rest of the paper is organized as follows. Sect. 2 contains some preliminaries about TRS. Sect. 3 is the technical core of the paper, where our semantics is technically defined and many strong properties are proved: polarity, compositionality, adequacy wrt rewriting. In Sect. 4 we discuss in detail the question of full abstraction. Finally Sect. 5 presents some conclusions. Detailed proofs can be found in Appendix A.

2 Preliminaries

We assume a first order signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint sets of *constructor* and *function* symbols resp., all them with associated arity. We write DC^n (FS^n resp.) for the set of constructor (function) symbols of arity n, and also Σ^n for any symbol of the signature of arity n. We also assume a numerable set $\mathcal{V}$ of variables. As usual notations we write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables. The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in \Sigma^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to DC^n (so $CTerm \subseteq Exp$).[1] We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects along the paper.

We consider also the extended signature $\Sigma_\bot = \Sigma \cup \{\bot\}$, where $\bot$ is a new 0-arity constructor symbol that stands for the undefined value. Over this signature we define the sets $Exp_\bot$ and $CTerm_\bot$ of *partial* expressions and c-terms resp. The intended meaning is that Exp and $Exp_\bot$ stand for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ and $CTerm_\bot$ stand for data terms representing total and partial values resp.

[1] We use the terminology Exp (for general expressions) instead of the more usual $Term$ in order to highlight the syntactic (and semantic) difference with $CTerm$ (data values).

The *shell* $|e|$ of an expression e represents the outer constructed part of e and is defined as: $|X| = X$; $|c(e_1,\ldots,e_n) = c(|e_1|,\ldots,|e_n|)$; $|f(e_1,\ldots,e_n)| = \bot$.

Substitutions $\theta \in Subst$ are mappings $\theta : \mathcal{V} \to Exp$, extending naturally to $\theta : Exp \to Exp$.

One-hole contexts are defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1,\ldots,\mathcal{C},\ldots,e_n)$, with $h \in \Sigma^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1,\ldots,\mathcal{C},\ldots,e_n)[e] = h(e_1,\ldots,\mathcal{C}[e],\ldots,e_n)$.

The approximation ordering $\sqsubseteq$ is defined on expressions as the least partial ordering satisfying: *i)* $\bot \sqsubseteq e$ for all $e \in Exp_\bot$, and *ii)* $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$.

A *constructor-based term rewriting system* $\mathcal{P}$ (*CS*, also called *program* along this paper) is a set of rewrite rules of the form $f(\bar{t}) \to e$ where $f \in FS^n$, $e \in Exp$, $var(e) \subseteq var(\bar{t})$, and $\bar{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\bar{t}$. Given a program $\mathcal{P}$, its associated rewrite relation $\to_\mathcal{P}$ is defined as: $\mathcal{C}[l\theta] \to_\mathcal{P} \mathcal{C}[r\theta]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\theta \in Subst$. We write $\overset{*}{\to}_\mathcal{P}$ for the reflexive and transitive closure of the relation $\to_\mathcal{P}$. In the following, we will usually omit the reference to $\mathcal{P}$.

3 A semantics for CS

In this section we present our proposed semantics, which has a logic flavor as it is based on a proof calculus. The use of proof calculi to specify the semantics of rewriting formalisms is not unfrequent. Two well-known cases correspond to the frameworks of rewriting logic [14] and CRWL [9]. We have been inspired by the philosophy of the latter, according to the following roadmap:

– We first identify the 'finite pieces' of which the denotation of expressions should be made of. In our case these will be the *s-cterms* introduced in Sect. 3.1, capturing technically the idea of 'packaging sets below constructor' mentioned in Sect. 1.

– Then, we devise a proof calculus able to prove statements of the form $e \twoheadrightarrow st$ expressing that st is a finite approximation to the denotation of e. This will be done in Sect. 3.2, although technically expressions will be generalized to the more general *s-expressions*.

– The proof calculus induces a natural notion of denotation of expression: it is simply the set of its provable approximations. The fact of working with finite approximations makes unnecessary to use a background of cpo's and powerdomains. This was found greatly convenient in the CRWL framework, and it is even more so in our case, where recursive nestings of constructors and sets occur. Moreover, it is known ([5, 15]) that an approach based on semantic domains with infinite (limit) elements and using fixpoint techniques has technical limitations ([5, 15]).

– If the proof calculus is designed to have a 'compositional' aspect, then one can expect compositionality of the resulting semantics, and the proof calculus is in itself a great aid to prove it. We have pursued this design principle in our proof calculus; as a result, and we have been able to prove compositionality and other relevant properties of the semantics (Sect. 3.2).

– Now, since or aim is to develop a new semantics for standard rewriting, not to give a new notion of rewriting, it is essential to show that our semantics is indeed related to rewriting: this is done in Sect. 3.3 by correctness and completeness results.

– Finally, with all the previous results and an extra little effort, we are able to prove full abstraction of our semantics (Sect. 4).

3.1 SCTerms: the pieces of the semantics

In this section we define new syntactic notions (of expressions, cterms, etc) in order to pack different values coming from non deterministic reductions at the syntactic level, by introducing sets in the corresponding syntax. Values become *s-cterms* that must be defined in mutual recursion with *elemental s-cterms*:

$$\begin{aligned} &ESCTerm \ni est ::= X \mid c(st_1,\ldots,st_n) \\ &\qquad \text{for } X \in \mathcal{V}, c \in DC^n, st_1,\ldots,st_n \in SCTerm \\ &SCTerm \ni st ::= \emptyset \mid \{est_1,\ldots,est_n\} \\ &\qquad \text{for } n > 0, est_1,\ldots,est_n \in ESCTerm \end{aligned}$$

A s-cterm is a *finite* set of elemental s-cterms, that are variables or constructors applied to s-cterms. The aim of these values is to capture the reduction of a non deterministic expression like $c(amb(a,b))$ into the single value

$\{c(\{a,b\})\}$. With the same idea, but allowing function symbols we define *elemental s-expressions* and *s-expressions* as:

$$\begin{array}{l} ESExp \ni ese ::= X \mid h(se_1,\ldots,se_n) \\ \qquad \text{for } X \in \mathcal{V}, h \in \Sigma^n, se_1,\ldots,se_n \in SExp \\ SExp \ni se ::= \emptyset \mid \{ese_1,\ldots,ese_n\} \\ \qquad \text{for } n > 0, ese_1,\ldots,ese_n \in ESExp \end{array}$$

In the definition of $SCTerm$ (and also in $SExp$), the base case $\emptyset$ could be hidden in the brace notation $\{est_1,\ldots,est_n\}$ just permitting $n = 0$ (in fact, we will do it sometimes). We have preferred to emphasize the presence of $\emptyset$, which plays the role of the undefined value (similar to $\perp$ for Exp in Sect. 2). Therefore s-cterms and s-expressions should be understood as *partial*. *Total* s-expressions and s-cterms would not use $\emptyset$, but they do not play any significant role in the following.

We can flatten a s-expression se to obtain the set $flat(e)$ of expressions "contained" in it: $flat(\emptyset) = \{\perp\}$ and $flat(se) = \bigcup_{ese \in se} flat(ese)$ if $se \neq \emptyset$, where $flat$ for elemental s-expressions is defined as $flat(X) = \{X\}$; $flat(h(se_1,\ldots,se_n)) = \{h(e_1,\ldots,e_n) | e_i \in flat(se_i) \text{ for } i = 1..n\}$.

The set $SSubst$ of *s-substitutions* consists of mappings $\sigma : \mathcal{V} \to SExp$. The *domain* of a s-substitution σ is defined as $dom(\sigma) = \{X \mid \sigma(X) \neq \{X\}\}\}$. Notice that s-substitutions replace variables by s-expressions (which are sets), and some care must be taken when extending s-substitutions to $eSExp$ and $SExp$:

$$\begin{array}{ll} \sigma : eSExp \to SExp & \sigma : SExp \to SExp \\ X\sigma = \sigma(X) & \{ese_1,\ldots,ese_n\}\sigma = \bigcup_{i\in\{1..n\}} ese_i\sigma \\ h(\overline{se})\sigma = \{h(\overline{se\sigma})\} & \end{array}$$

The set $SCSubst$ of *s-csubstitutions* consists of of mappings $\sigma : \mathcal{V} \to SCTerm$ and the extensions to the domains of $eSCTerm$ and $SCTerm$ are defined analogously to the previous extensions.

One hole (elemental) *s-contexts* are defined as:

$$sCntxt \ni s\mathcal{C} ::= [\,] \mid \{\ldots, h(\ldots, s\mathcal{C}, \ldots), \ldots\} \quad \text{with } h \in \Sigma \text{ and } s\mathcal{C} \in sCntxt$$

The application of a context to a s-expression is defined in the natural way. Notice that s-contexts allow only the hole to be in the place of a sub-s-expression. For example, the possible s-contexts of $\{Y, c(\{X\})\}$ are $[\,]$ and $\{Y, c([\,])\}$, but not $\{[], c(\{X\})\}$ neither $\{Y, []\}$.

The preorder $\sqsubseteq$ is defined for s-expressions as the least preorder satisfying: $se \sqsubseteq se'$ if $\forall ese \in se.\exists ese' \in se'$ such that $ese \sqsubseteq ese'$, where for elemental s-expressions $\sqsubseteq$ is defined as the least preorder such that: $X \sqsubseteq X$ for any $X \in \mathcal{V}$ and $h(se_1,\ldots,se_n) \sqsubseteq h(se'_1,\ldots,se'_n)$ iff $se_i \sqsubseteq se'_i$ for $i = 1..n$. For s-substitutions, the preorder is defined as $\sigma \sqsubseteq \sigma'$ if $\sigma(X) \sqsubseteq \sigma'(X)$ for $X \in \mathcal{V}$.

Programs are exactly those defined in Sect. 3. The proof calculus of the next section needs to use function rules transformed into the new syntactical framework of s-expressions. For this purpose we define the transformation of $e \in Exp$ into a s-expression $\tilde{e} \in SExp$ as: $\tilde{\perp} = \emptyset$; $\tilde{X} = \{X\}$ for any $X \in \mathcal{V}$; $\widetilde{h(e_1,\ldots,e_n)} = \{h(\tilde{e_1},\ldots,\tilde{e_n})\}$, with $h \in \Sigma^n$. The transformation $\tilde{\mathcal{C}}$ of a context $\mathcal{C}$ is defined in the natural way and its application to a s-expression is defined as $\tilde{\mathcal{C}}[e] = \tilde{\mathcal{C}}[\tilde{e}]$. On the other hand, $\tilde{\sigma}$ is defined as $\tilde{\sigma}(X) = \widetilde{\sigma(X)}$, for $\sigma \in Subst$.

3.2 A Proof Calculus

Our goal in this section is to devise a proof calculus to specify which $SCTerm$'s correspond to a given expression under a given CS. To do that we will inspire in the *CRWL* proof calculus [9], adapting it to serve our purposes. Therefore the expressions will be evaluated in an innermost way and we will avoid the use of any transitivity rule, in order for the calculus to be compositional in the values it computes. But as we will use partial s-cterms as values then this innermost evaluation will not induce the strictness of functions, hence enabling the completeness of our semantics wrt term rewriting even for non-terminating CS.
Besides, during parameter passing the variables in the rewrite rules will be instanciated with partial s-cterms. As a consequence it is possible to end up evaluating expressions with some $SCTerm$ "inside" (as a subexpression), even when starting the computation from an ordinary $e \in Exp$. So, instead of dealing only with expressions from Exp, our calculus will compute the partial s-cterms corresponding to any given partial s-expression. Finally, the mapping $\tilde{}$ will be used in combination with our logic to get the $SCTerm$'s corresponding to a given Exp.

E	$se \twoheadrightarrow \emptyset$	
RR	$\{X\} \twoheadrightarrow \{X\}$	if $X \in \mathcal{V}$
DC	$\dfrac{se_1 \twoheadrightarrow st_1 \ \ldots \ se_n \twoheadrightarrow st_n}{\{c(se_1,\ldots,se_n)\} \twoheadrightarrow \{c(st_1,\ldots,st_n)\}}$	if $c \in CS$
More	$\dfrac{se \twoheadrightarrow st_1 \ \ldots se \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \ldots \cup st_n}$	
Less	$\dfrac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots \ \{esa_m\} \twoheadrightarrow st_m}{\{ese_1,\ldots,ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m}$	if $n \geq 2, m > 0$, for any $\{esa_1,\ldots,esa_m\} \subseteq \{ese_1,\ldots,ese_n\}$
ROR	$\dfrac{se_1 \twoheadrightarrow \tilde{p_1}\theta \ \ldots \ se_n \twoheadrightarrow \tilde{p_n}\theta \ \ \tilde{r}\theta \twoheadrightarrow st}{\{f(se_1,\ldots,se_n)\} \twoheadrightarrow st}$	if $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$, $\theta \in SCSubst$

Fig. 1. A proof calculus for constructor systems

To be precise, our proof calculus will prove reduction statements of the form $se \twoheadrightarrow st$ with $se \in SExp$ and $st \in SCTerm$, expressing that st represents an approximation to one of the possible structured sets of values for se. This calculus is presented in Fig. 1. Rule E (empty) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) work with singleton sets and allow us to reduce any variable to itself, and to decompose the evaluation of a constructor-rooted elemental s-expression. Rule More allows us to compute more than one value for an s-expression, and to collect these values together. Rule Less allows us to discard some elemental s-expressions from the s-expression under evaluation. Finally rule ROR (run-time[2] outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of a $SCSubst\ \theta$) and then reduce the instantiated right-hand side. The use of $SCSubst$'s is fundamental to get the exact behaviour of term rewriting, because then the branching information associated to the computation of each $\tilde{p_i}\theta$ is not lost in some kind of flattening to a set of c-terms, but kept into the structured representation of $SCTerm$'s.

We write $\mathcal{P} \vdash se \twoheadrightarrow st$ to express that $se \twoheadrightarrow st$ is derivable in our calculus under the CS $\mathcal{P}$. The *denotation* of a s-expression se under a CS $\mathcal{P}$ is defined as $[\![se]\!]^{\mathcal{P}} = \{st \in SCTerm \mid \mathcal{P} \vdash se \twoheadrightarrow st\}$. In the following we will usually omit the reference to $\mathcal{P}$.

Example 2. Consider the CS of Ex. 1, we can use our calculus to prove the statement $f(c(\widetilde{amb(a,b)})) \twoheadrightarrow \widetilde{d(a,b)}$ (some steps have been omitted for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{a\} \twoheadrightarrow \{a\}}\ \text{DC}\ \dfrac{}{\{b\} \twoheadrightarrow \emptyset}\ \text{E}\ \dfrac{\ldots}{\{a\} \twoheadrightarrow \{a\}}}{\{amb(\{a\},\{b\})\} \twoheadrightarrow \{a\}}\ \text{ROR}\ \dfrac{\ldots}{\{amb(\{a\},\{b\})\} \twoheadrightarrow \{b\}}\ \text{ROR}}{\{amb(\{a\},\{b\})\} \twoheadrightarrow \{a,b\}}\ \textsc{More}}{\{c(\{amb(\{a\},\{b\})\})\} \twoheadrightarrow \{c(\{a,b\})\}}\ \text{DC}\ \dfrac{(*)}{\{d(\{a,b\},\{a,b\})\} \twoheadrightarrow \{d(\{a\},\{b\})\}}\ \text{DC}}{f(c(\widetilde{amb(a,b)})) \equiv \{f(\{c(\{amb(\{a\},\{b\})\})\})\} \twoheadrightarrow \{d(\{a\},\{b\})\} \equiv \widetilde{d(a,b)}}\ \text{ROR}$$

where $(*)$ is the derivation:

$$\dfrac{\dfrac{\dfrac{}{\{a\} \twoheadrightarrow \{a\}}\ \text{DC}}{\{a,b\} \twoheadrightarrow \{a\}}\ \textsc{Less}\ \dfrac{\ldots}{\{a,b\} \twoheadrightarrow \{b\}}}{\{d(\{a,b\},\{a,b\})\} \twoheadrightarrow \{d(\{a\},\{b\})\}}\ \text{DC}$$

On the other hand, $\widetilde{d(a,b)}$ is not a correct value for $f(\widetilde{amb(c(a),c(b))})$, because in that expression the evaluation of $amb(\widetilde{c(a)},c(b))$ has to be performed in order to get a value matching the argument of the left-hand side of the only rule for f, and the only matching values for it are $\widetilde{c(a)}$, $\widetilde{c(b)}$ and $\{c(\emptyset)\}$, as for example $\{c(\{a\}),c(\{b\})\}$ does not match $\widetilde{c(X)}$.

Notice that, structurally, a denotation $[\![se]\!]$ is a possibly infinite set of s-cterms, each one being a finite set of elemental s-cterms. Infinite denotations might appear with non-terminating programs. Notice, however, that the

[2] The prefix 'run-time' comes from 'run-time choice', which is often applied ([12, 9]) to the parameter passing mechanism of term rewriting.

elements are s-cterms that are, by construction, finite objects. Thus, we avoid the presence of infinite values as elements of denotations.

As we anticipated above, even when proving a reduction $\tilde{e} \twoheadrightarrow \tilde{t}$ for $e \in Exp$ and $t \in CTerm$, we may find premises of the shape $se \twoheadrightarrow st$ for $se \in SExp, st \in SCTerm$, because the substitutions used for parameter passing in ROR may introduce sets in $\tilde{r}\theta$, as we can see in the second premise of the first application of ROR, in Ex. 2. But in fact the kind of s-expressions that we may find in the proof for some reduction for an expression is more restricted. It is easy to prove that in any proof for any statement $\tilde{e} \twoheadrightarrow st$ with $e \in Exp$ we have that in any premise $se' \twoheadrightarrow st'$ for it, $se \in trSExp$, a set defined as $trSExp \ni tr ::= st \mid \{h(tr_1, \ldots, tr_n)\}$, with $st \in SCTerm, h \in \Sigma, tr_1, \ldots, tr_n \in trSExp$.
This suggests that we could have defined our logic to prove reductions $se \twoheadrightarrow st$ with $se \in trSExp$ and $st \in SCTerm$ only, but we think that it is more profitable to define it to deal with the more general case of $se \in SExp$. First of all, then we get a logic that can handle a more general kind of syntactic objects, and therefore that could be used to express other formalism apart from term rewriting. We could slightly modify the rule ROR to accept not only CS's but in general *"s-expression rewriting systems" (sCS's)*, consisting of rules $\{f(st_1, \ldots, st_n)\} \twoheadrightarrow se$. This way the original formulation of ROR becomes a particular case of the new version, that works with CS's adapted to sCS's by means of $\tilde{\ }$. This would be similar to what is done in [16] to express term rewriting, term graph rewriting and noncopying rewriting by means of the more general framework of marked term rewriting. We consider this an interesting possible subject of future work.

On the other hand, working with reductions of s-expressions allows us to formulate more general and powerful results about the semantics, which become easier to prove because of its generality (which gives us stronger induction hypotheses), and that can be then easily applied to the more restricted case. These are nice properties like the following *polarity* property of our semantics. In all the results of this and next section we assume a given CS and omit mentioning it.

Proposition 1 (Polarity). *Let $se, se' \in SExp$, $st, st' \in SCTerm$. If $se \sqsubseteq se'$ and $st' \sqsubseteq st$ then $st \in [\![se]\!]$ implies $st' \in [\![se']\!]$.*

Our semantics also enjoys the following monotonicity property related to substitutions. It is formulated for the preorder $\sqsubseteq$ and for the preorder $\trianglelefteq$ over $SSubst$, defined by $\sigma \trianglelefteq \sigma'$ iff $\forall X \in \mathcal{V}, [\![\sigma(X)]\!] \subseteq [\![\sigma'(X)]\!]$.

Proposition 2 (Monotonicity of substitutions). *Let $se \in SExp$, $\sigma, \sigma' \in SSubst$. If $\sigma \trianglelefteq \sigma'$ or $\sigma \sqsubseteq \sigma'$ then $[\![se\sigma]\!] \subseteq [\![se\sigma']\!]$.*

One of the most important properties of our logic is its *compositionality*, a property very close to the DET-additivity property for algebraic specifications of [12], which will be one of the keys for full abstraction.

Theorem 1 (Compositionality). *For all $s\mathcal{C} \in sCntxt, se \in SExp$,*

$$[\![s\mathcal{C}[se]]\!] = \bigcup_{st \in [\![se]\!]} [\![s\mathcal{C}[st]]\!]$$

As a consequence: $[\![se]\!] = [\![se']\!] \Leftrightarrow \forall s\mathcal{C}.\ [\![s\mathcal{C}[se]]\!] = [\![s\mathcal{C}[se']]\!]$.

Regarding *closedness* under susbtitutions, as we use $SCSubst$ for parameter passing it is natural to have closedness of reductions under this type of substitutions. Besides, as rewriting is closed under $Subst$ it is expected to have some kind of closedness for $Subst$ too. But in general it is not true that for any $st \in SCTerm, \sigma \in SSubst$ we have $st\sigma \in SCTerm$, therefore it makes no sense to expect that $se \twoheadrightarrow st$ implies $se\sigma \twoheadrightarrow st\sigma$, as the reductions in our logic are from $SExp$ to $SCTerm$. Nevertheless we still can say something about that, as we can see in the following property.

Proposition 3 (Closedness under substitutions). *Let $se \in SExp$, $st \in SCTerm$. If $st \in [\![se]\!]$ then: a) $\forall \theta \in SCSubst, st\theta \in [\![se\theta]\!]$ b) $\forall \sigma \in SSubst, [\![st\sigma]\!] \subseteq [\![se\sigma]\!]$*

All these properties are powerful tools to reason about the denotations of s-expression. And this reasoning power is transfered to the term rewriting universe through the adequacy results that we will see in the next section, thus opening paths for the development of new reasoning techniques for constructor systems.

$\_ \vdash \_ \lessdot \_ \subseteq CS \times SCTerm \times Exp$	$\_ \vdash \_ \lessdot \_ \subseteq CS \times ESCTerm \times Exp$
$\mathcal{P} \vdash st \lessdot e$ if $\forall est \in st, \mathcal{P} \vdash est \lessdot e$	$\mathcal{P} \vdash X \lessdot e$ if $\mathcal{P} \vdash e \to^* X$
	$\mathcal{P} \vdash c(\overline{st}) \lessdot e$ if $\mathcal{P} \vdash e \to^* c(\overline{e})$ for some $\overline{e}$ such that $\forall e_i \in \overline{e}, \mathcal{P} \vdash st_i \lessdot e_i$

Fig. 2. Domination relation

3.3 Relation with rewriting

The nice properties of our logic could reveal pretty useless if not accompanied by strong adequacy results that relate this logic to the term rewriting relation. In the present section we will formally state and prove these results.

The keys to prove this adequacy are the following lemmas, whose meaning will be clarified later on.

Lemma 1. *Let $\sigma \in SSubst, se \in SExp, st \in SCTerm$. If $se\sigma \twoheadrightarrow st$ then there exists $\theta \in [\![\sigma]\!]$ such that $se\theta \twoheadrightarrow st$.*

Lemma 2. *Let $e \in Exp, st \in SCTerm, \theta \in SCSubst$. If $\tilde{e}\theta \twoheadrightarrow st$ then $st \lessdot e\sigma$ for any $\sigma \in Subst$ such that $\theta \lessdot \sigma$.*

First of all we want our logic to be *complete*, that for any expression our semantics could capture any c-term reachable from it by rewriting. This is the first result we get about that:

Proposition 4. *For all $e, e' \in Exp$, if $e \to^* e'$ then $[\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$.*

We can prove it for one step (for $e \to e'$) and then just generalize it to many steps by induction on the length of $e \to^* e'$. The keys are the compositionality of Theorem 1, and Lemma 1, which expresses that in any reduction $se\sigma \twoheadrightarrow st$ only a finite amount of the information contained in σ is needed. We formalize it through the notion of denotation of a $SSubst$, defined as $[\![\sigma]\!] = \{\theta \in SCSubst \mid \forall X \in \mathcal{V}, \sigma(X) \twoheadrightarrow \theta(X)\}$. We can use these tools as follows:

Proof (Sketch). Assume $e \to e'$, if the step was performed at the root then we have $e \equiv f(\overline{p})\sigma \to r\sigma \equiv e'$ for some rule $(f(\overline{p}) \to r) \in \mathcal{P}$. Now assume $\widetilde{r\sigma} \twoheadrightarrow st$, then as $\widetilde{r\sigma} \equiv \tilde{r}\tilde{\sigma}$, by Lemma 1 $\exists \theta \in [\![\tilde{\sigma}]\!]$ such that $\tilde{r}\theta \twoheadrightarrow st$, but then it is easy to prove that $\widetilde{f(\overline{p})}\theta \twoheadrightarrow st$. Besides $\theta \in [\![\tilde{\sigma}]\!]$ implies $\theta \trianglelefteq \tilde{\sigma}$, and so we can apply Prop. 2 to get $\widetilde{f(\overline{p})\sigma} \equiv \widetilde{f(\overline{p})}\tilde{\sigma} \twoheadrightarrow st$.
If the step was not performed at the root then we have $e \equiv \mathcal{C}[f(\overline{p})\sigma] \to \mathcal{C}[r\sigma] \equiv e'$, and so we can combine the result for the previous case with the compositionality of Theorem 1 to get the desired result.

Now we can apply Prop. 4 to get the following strong completeness result.

Theorem 2 (Completeness). *For all $e, e' \in Exp, t \in CTerm$,*
a) $e \to^* e'$ implies $\widetilde{|e'|} \in [\![\tilde{e}]\!]$ *b) $e \to^* t$ implies $\widetilde{t} \in [\![\tilde{e}]\!]$*

Proof. Concerning *a)*, it is easy to prove that $\forall e \in Exp, \tilde{e} \twoheadrightarrow \widetilde{|e|}$, by induction on the structure of Exp. But then $\widetilde{|e'|} \in [\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$ by Prop. 4. Concerning *b)*, we can prove $\forall t \in CTerm, |t| \equiv t$ by induction on $CTerm$, and so $\tilde{e} \twoheadrightarrow \widetilde{|t|} \equiv \widetilde{t}$, by *a)*.

We also want our logic to be *correct*, that for any expression our semantics could not compute more c-terms than those reachable by rewriting. One key ingredient will be the *domination relation* $\_ \lessdot \_$ defined in Fig. 2 (we will omit the prefix "$\vdash \mathcal{P}$" when it is implied by the context). With this relation we try to transfer to the rewriting world the finer distinction between sets of values that the structured representation of $SCTerm$ allows us to perform. This way under the CS of Ex. 1 we have $\{c(\{0,1\})\} \lessdot c(amb(0,1))$ but not $\{c(\{0,1\})\} \lessdot amb(c(0), c(1))$. The domination relation $\_ \lessdot \_$ has a strong relation with our semantics, as stated in the following result:

Lemma 3 (Domination). *For all $e \in Exp, st \in SCTerm$: $st \in [\![\tilde{e}]\!]$ iff $st \lessdot e$.*

But notice that $\_ \lessdot \_$ only talks about reductions for $\tilde{e}$ with $e \in Exp$, and so it cannot be used to formulate properties like those seen in Sect. 3.2, although it inherits them through Lemma 3.
The key to prove Lemma 3 is Lemma 2, in which we extend the relation $\_ \lessdot \_$ to $\_ \vdash \_ \lessdot \_ \subseteq CS \times SCSubst \times Subst$ by $\theta \lessdot \sigma$ iff $\forall X \in \mathcal{V}, \theta(X) \lessdot \sigma(X)$. Lemma 2 expresses that given a reduction $\tilde{e}\theta \twoheadrightarrow st$ then any substitution σ that contains at least the same information as θ can be used to dominate st. Note we have also used the domination to encode this "containment of at least the same information", hence, as $\_ \lessdot \_$ is a rewriting-based notion, we have been able to change from the universe of our logic to the universe of rewriting along the way.

Proof (For Lemma 3, sketch). The left to right direction is a trival corollary of Lemma 2, just taking $\theta = \epsilon = \sigma$. The converse implication follows from a simple induction on the proof for $st \lessdot e$, using the completeness of our logic as stated in Prop. 4.

The good thing about $\_ \lessdot \_$ is that it already has a strong conection with rewriting, as it is defined by means of rewriting derivations. Hence we can perform a simple induction on the structure of $SCTerm$ and $ESCTerm$ to prove the following result, which uses the notion of flattening defined in Sect. 3.1.

Lemma 4. *Let $st \in SCTerm$, $est \in ESCTerm$, $e \in Exp$, and assume $t \in flat(st)$. If $st \lessdot e$ then $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.*

And now we are ready to state and prove our main correctness result.

Theorem 3 (Correctness). *Let $e \in Exp, st \in SCTerm, t \in CTerm_\perp$:*

a) If $st \in [\![\tilde{e}]\!]$ and $t \in flat(st)$, then $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.
b) If $\tilde{t} \in [\![\tilde{e}]\!]$, then $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.
c) Besides, in a) or b), if $t \in CTerm$, then $e \to^* t$.

Proof. We get *a)* just chaining Lemma 3 and Lemma 4. Concerning *b)*, we can prove that $\forall t \in CTerm_\perp, flat(\tilde{t}) = \{t\}$ by induction on $CTerm_\perp$, and chain it with *a)*. Finally *c)* is a consequence of *a)* and *b)*, because if $t \in CTerm$ then it is maximal wrt $\sqsubseteq$, hence $t \sqsubseteq |e'|$ implies $t \equiv |e'|$. But that implies there is no $\perp$ in $|e'|$, therefore $e' \in CTerm$ and $e' \equiv |e'| \equiv t$, and so $e \to^* e' \equiv t$.

4 Full abstraction

The semantics $[\![se]\!]^{\mathcal{P}}$ of Sect. 3 is defined for s-expressions, but induces naturally a notion of semantics for ordinary expressions $e \in Exp$:

$$[\![e]\!]^{\mathcal{P}}_S = [\![\tilde{e}]\!]^{\mathcal{P}}(= \{st \in SCTerm | \tilde{e} \twoheadrightarrow st\})$$

In this section we discuss full abstraction in the context of CS and show that $[\![\_]\!]_S$ achieves it, in contrast to semantics directly based on sets of results, informally described in Sect. 1.

The property of *full abstraction* expresses a perfect capture of observational behavior: in a fully abstract semantics, two expressions have the same semantics if and only if they are observationally indistinguishable. For this to be meaningful, one must choose a criterion of observability. The problem of full abstraction was first investigated by Plotkin [17] in connection to PCF (a simple functional language), and since then is a standard topic when dealing with program semantics (see e.g. [18]). It is common to adjust its definition to the characteristics of the language under consideration. In the context of functional-like programming languages like PCF the condition for full abstraction is usually stated as:

$$[\![e]\!] = [\![e']\!] \Leftrightarrow \mathcal{O}(\mathcal{C}[e]) = \mathcal{O}(\mathcal{C}[e']), \text{for any context } \mathcal{C} \tag{1}$$

where $\mathcal{O}$ is the observation function of interest. Programs do not need to be mentioned, because programs and expressions can be identified by contemplating the evaluation of e under $\mathcal{P}$ as the evaluation of a big λ-expression or big *let*-expression embodying $\mathcal{P}$ and e. Contexts pose no problems either. In our case, since programs (CS) are kept different from expressions, some care must be taken. It might happen that $\mathcal{P}$ has not enough syntactical elements and rules to built interesting distinguishing contexts. For instance, if in Ex. 1 we drop the definition of f, then we cannot built a context that distinguishes $c(a?b)$ from $c(a)?c(b)$. This would imply that soundness or full abstraction would not be intrinsic to the semantics, but would greatly depend on the program. What we need is requiring the right part of (1) to hold for all contexts that might be obtained by extending $\mathcal{P}$ with new auxiliary functions. To be more precise, we say that $\mathcal{P}'$ is a *safe extension* of $(\mathcal{P}, e)$ if $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}''$, where $\mathcal{P}''$ does not include defining rules for any function symbol occurring in $\mathcal{P}$ or e. Any sensible notion of semantics should verify $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$ when $\mathcal{P}'$ safely extends (P, e). This happens indeed for all the semantics considered below.

Things are now prepared to give the following definition:

Definition 1 (Observations, full abstraction).
(a) A semantic function (a semantics*, in short) is a function* $[\![\text{-}]\!]$ *associating a semantic value (taken from a set* $\mathcal{D}$*) to each expression* e *under a given program* $\mathcal{P}$*. We write* $[\![e]\!]^{\mathcal{P}}$ *for such value.*
(b) An observation function *is a function* $\mathcal{O}$ *associating a set of observation values (or* observables*, taken from a set Obs) to each expression* e *under a given program* $\mathcal{P}$*. We write* $\mathcal{O}^{\mathcal{P}}(e)$ *for it.*
(c) A semantics is fully abstract *wrt* $\mathcal{O}$ *iff for any* $\mathcal{P}$ *and* $e, e' \in Expr$*, the following two conditions are equivalent:*
(i) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ *(ii)* $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$ *for any* $\mathcal{P}'$ *safely extending* $(\mathcal{P}, e)$*,* $(\mathcal{P}, e')$ *and any* $\mathcal{C}$ *built with the signature of* $\mathcal{P}'$*.*
In words: semantic equality is equivalent to observational indistinguishability.
(d) A notion weaker than full abstraction is: a semantics is sound *wrt* $\mathcal{O}$ *iff the condition* (i) *above implies the condition* (ii)*.*
In words: semantic equality implies observational indistinguishability.
(e) A semantics is compositional *iff for any* $\mathcal{P}$ *and* $e, e' \in Expr$*, the following two conditions are equivalent:*
(i) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ *(ii)* $[\![\mathcal{C}[e]]\!]^{\mathcal{P}} = [\![\mathcal{C}[e']]\!]^{\mathcal{P}}$ *for any* $\mathcal{C}$ *built with the signature of* $\mathcal{P}$*.*
In words: the semantics of an expression depends only on the semantics of its subexpressions. Notice that $(ii) \Rightarrow (i)$ *holds trivially (take* $\mathcal{C} = [\ ]$*).*

In the next definition we collect some notions of semantics and observables for the case of CS. $[\![\text{-}]\!]_S$ and $[\![\text{-}]\!]_{S'}$ are our new contributed semantics; the rest are the 'obvious' semantics of Sect. 1. As usual, we omit the program $\mathcal{P}$ in notations.

Definition 2 (Semantics and observations for CSs).
We consider the following semantics for expressions $e \in Expr$*:*

$[\![e]\!]_{rw} = \{e' \mid e \to^* e'\}$ $\quad$ $[\![e]\!]_{nf} = \{e' \mid e \to^* e', e' \text{ in normal form}\}$
$[\![e]\!]_t = \{t \in CTerm \mid e \to^* t\}$ $\quad$ $[\![e]\!]_{t_\perp} = \{t \in CTerm_\perp \mid \exists e'.(e \to^* e' \wedge t \sqsubseteq |e'|)\}$
$[\![e]\!]_S = [\![\tilde{e}]\!]$ $\quad$ $[\![e]\!]_{S'} = \bigcup_{st \in [\![e]\!]_S} st.$

We consider the following observation functions for expressions $e \in Expr$*:*

$\mathcal{O}_t(e) = [\![e]\!]_t$ $\quad$ $\mathcal{O}_{t_\perp}(e) = [\![e]\!]_{t_\perp}$ $\quad$ $\mathcal{O}_{true}(e) = [\![e]\!]_t \cap \{true\}$

Some remarks:
• It is clear that $[\![e]\!]_t \subseteq [\![e]\!]_{nf} \subseteq [\![e]\!]_{rw}$, and also $[\![e]\!]_t \subseteq [\![e]\!]_{t_\perp}$.
• Notice that some of the sets above can play at the same time the role of semantic values and of observation values.
• $[\![e]\!]_S$ was introduced at the beginning of the this section. $[\![e]\!]_{S'}$ is a simplified variant, making more readable the semantics of particular expressions, because $[\![e]\!]_S$ is a set of finite sets of $est's$, while $[\![e]\!]_{S'}$ is simply a set of $est's$. However $[\![\text{-}]\!]_S$ has been technically more convenient for proving properties of the semantics, due to its more direct connection to the proof calculus. Both semantics are essentially the same, as evidenced by:

Proposition 5. *For any* $e, e' \in Exp$*,* $[\![e]\!]_S = [\![e']\!]_S \Leftrightarrow [\![e]\!]_{S'} = [\![e']\!]_{S'}$

The next result shows that, although $\mathcal{O}_t, \mathcal{O}_{t_\perp}, \mathcal{O}_{true}$ define different observations, it is irrelevant which is chosen, as far as full abstraction is concerned.

Proposition 6. *Assume a given semantics* $[\![\text{-}]\!]$*. Then:*
(a) $[\![\text{-}]\!]$ *is fully abstract wrt* $\mathcal{O}_t \Leftrightarrow [\![\text{-}]\!]$ *is fully abstract wrt* $\mathcal{O}_{t_\perp}$*.*
(b) If expressions to be observed are restricted to be ground, then:
$[\![\text{-}]\!]$ *is fully abstract wrt* $\mathcal{O}_t \Leftrightarrow$ *is fully abstract wrt* $\mathcal{O}_{t_\perp} \Leftrightarrow$ *is fully abstract wrt* $\mathcal{O}_{true}$

The groundness condition is necessary in *(b)*, as sketchly discussed here: in Th. 4 below we prove that $[\![\text{-}]\!]_S$ is fully abstract wrt $\mathcal{O}_t$. However, it is not fully abstract wrt $\mathcal{O}_{true}$ if non-ground expressions are considered: for instance, $[\![X]\!]_S = \{\emptyset, \{X\}\} \neq [\![Y]\!]_S$. However, it can be shown (left linearity is essential here) that for any $\mathcal{C}$, $\mathcal{C}[X] \to^* true \Rightarrow \mathcal{C}[Y] \to^* true$.

Now we show that the first four semantics, $[\![]\!]_{rw}$-, $[\![\text{-}]\!]_{nf}$, $[\![\text{-}]\!]_t$, $[\![\text{-}]\!]_{t_\perp}$ do not have good properties, as was informally discussed in Sect. 1. We use $\mathcal{O}_t$ in the result but, according to the previous result, $\mathcal{O}_{t_\perp}$ and $\mathcal{O}_{true}$ could be used instead (for ground expressions, in the latter case). This remark extends also to Th. 4 below.

Proposition 7.
(a) $[\![\text{-}]\!]_{rw}, [\![\text{-}]\!]_{nf}, [\![\text{-}]\!]_t, [\![\text{-}]\!]_{t_\perp}$ *are not fully abstract wrt* $\mathcal{O}_t$.
(b) Moreover, $[\![\text{-}]\!]_{nf}, [\![\text{-}]\!]_t, [\![\text{-}]\!]_{t_\perp}$ *are not compositional, nor sound wrt* $\mathcal{O}_t$.
(c) $[\![\text{-}]\!]_{nf}$ *(*$[\![\text{-}]\!]_t$ *resp.) remains not compositional nor sound wrt* $\mathcal{O}_t$ *even if programs are restricted to be confluent (confluent and terminating, resp.).*

Proof. (a) For $[\![\text{-}]\!]_{nf}, [\![\text{-}]\!]_t, [\![\text{-}]\!]_{t_\perp}$, (a) is implied by (b). For $[\![\text{-}]\!]_{rw}$, just add a new function $g(X) \to f(X)$ to Ex. 1. It is easy to see that $f(a)$ and $g(a)$ are contextually indistinguishable wrt $\mathcal{O}_t$, but $[\![f(a)]\!]_{rw} \neq [\![g(a)]\!]_{rw}$.
(b) Example 1 serves for all the three semantics.
(c) For $[\![\text{-}]\!]_{nf}$, consider the program $\{f \to f, g \to c(f), h(c(X)) \to a\}$. We have $[\![f]\!]_{nf} = [\![g]\!]_{nf} = \emptyset$, but $[\![h(f)]\!]_{nf} = \emptyset \neq \{a\} = [\![h(g)]\!]_{nf}$, proving at the same time not compositionality and unsoundness. For $[\![\text{-}]\!]_t$, replace the above program by $\{f \to h(a), g \to c(f), h(c(X)) \to a\}$.

Finally, we show that our semantics do not present those problems.

Theorem 4 (Compositionality and full abstraction of $[\![\text{-}]\!]_S$).
$[\![\text{-}]\!]_S$ *and* $[\![\text{-}]\!]_{S'}$ *are compositional, and fully abstract wrt* $\mathcal{O}_t$.

Proof. We prove the results for $[\![\text{-}]\!]_S$. For $[\![\text{-}]\!]_{S'}$, just use Prop. 5. Compositionality follows easily from definition of $[\![\text{-}]\!]_S$ and compositionality of $[\![\text{-}]\!]$ (Th. 1).
For full abstraction, let $\mathcal{P}$ be any CS, and $e, e' \in Expr$. We must prove:

$$[\![e]\!]^{\mathcal{P}}_S = [\![e']\!]^{\mathcal{P}}_S \Leftrightarrow \forall \mathcal{P}', \mathcal{C}\ .\ \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$$

where $\mathcal{P}'$ ranges over safe extensions of $(\mathcal{P}, e)$ and $(\mathcal{P}, e')$, and $\mathcal{C}$ over contexts built with the signature of $\mathcal{P}'$.
$\boxed{\Rightarrow}$ Assume $[\![e]\!]^{\mathcal{P}}_S = [\![e']\!]^{\mathcal{P}}_S$. Since $\mathcal{P}'$ is a safe extension, we know that $[\![e]\!]^{\mathcal{P}'}_S = [\![e']\!]^{\mathcal{P}'}_S$. We prove $\mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) \subseteq \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$ (the other inclusion is similar). Let $t \in \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e])$, which means $\mathcal{C}[e] \to^*_{\mathcal{P}'} t$. By Th. 2 we know $\tilde{t} \in [\![\mathcal{C}[e]]\!]^{\mathcal{P}'}_S = [\![\mathcal{C}[e']]\!]^{\mathcal{P}'}_S$, where the last equality is justified by compositionality. But then, since $t \in flat(\tilde{t})$, we have (by Th. 3) that $\mathcal{C}[e'] \to^*_{\mathcal{P}'} t$, that is, $t \in \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$, as desired.
For the other implication we need two auxiliary constructions enabling to build a context that distinguishes two expressions having different semantics:
• Given $st \in SCTerm$, we define a c-term $\widehat{st}$ 'mirroring' st as follows:

$$\widehat{\emptyset} = \langle\,\rangle_0 \qquad \widehat{\{X\}} = X\ (X \in \mathcal{V}) \qquad \widehat{\{c(\overline{st_i})\}} = c(\overline{\widehat{st_i}})$$
$$\widehat{\{est_1, \ldots, est_n\}} = \langle \widehat{\{est_1\}}, \ldots, \widehat{\{est_n\}} \rangle_n\ (n > 1)$$

where $\langle\text{-}\rangle_n$ $(n \geq 0)$ are new tuple-forming constructor symbols. It is assumed here that $SCTerm, ESCTerm$ are equipped with any standard ordering.
• Given $st \in SCTerm$, the program $\mathcal{P}_{st}$ defines new functions $f_{st}, \ldots$ as follows:

$$f_\emptyset(X) \to \langle\,\rangle_0 \qquad f_{\{X\}}(U) \to U\ (X \in \mathcal{V}) \qquad f_{\{c(\overline{st_i})\}}(c(\overline{U_i})) \to c(\overline{f_{st_i}(U_i)})$$
$$f_{\{est_1, \ldots, est_n\}}(U) \to \langle f_{\{est_1\}}(U), \ldots, f_{\{est_n\}}(U) \rangle_n\ (n > 1)$$

The roles of $\widehat{st}, \mathcal{P}_{st}$ are made clear by the following lemma:

Lemma 5. *For any* $\mathcal{P}, st, e$*:* $st \in [\![e]\!]^{\mathcal{P}}_S \Leftrightarrow f_{st}(e) \to^*_{\mathcal{P}'} \widehat{st}$*, where* $\mathcal{P}' \equiv \mathcal{P} \cup \mathcal{P}_{st}$.

We can now proceed with the proof of the pending implication.
$\boxed{\Leftarrow}$ Assume that $\mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) \subseteq \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$ for any safe extension $\mathcal{P}'$ and context $\mathcal{C}$. We prove $[\![e]\!]^{\mathcal{P}}_S \subseteq [\![e']\!]^{\mathcal{P}}_S$ (the other inclusion is similar). Let $st \in [\![e]\!]^{\mathcal{P}}_S$. Let $\mathcal{P}' \equiv \mathcal{P} \cup \mathcal{P}_{st}$, which is a safe extension of $(P, e), (P, e')$. Lemma 5 ensures $f_{st}(e) \to^*_{\mathcal{P}'} \widehat{st}$, which means $\widehat{st} \in \mathcal{O}_t^{\mathcal{P}'}(f_{st}(e))$. Now, since $\mathcal{P}'$ is a safe extension, observational equivalence of e, e' implies $\widehat{st} \in \mathcal{O}_t^{\mathcal{P}'}(f_{st}(e'))$, which means $f_{st}(e') \to^*_{\mathcal{P}'} \widehat{st}$. Again by Lemma 5, we conclude that $st \in [\![e']\!]^{\mathcal{P}}_S$, as desired.

5 Conclusions

In this paper we have provided a semantics for constructor based rewriting systems that is fully abstract with respect to natural notions of observation that extract the outer constructor part of outcomes as relevant information of computations. To the best of our knowledge, this is the first time that full abstraction has been achieved

for this class of programs and observations. Along the way to this result we have made some contributions: After noticing that 'obvious' semantics based directly on rewrite sequences lack compositionality, our main insight has been that it can be recovered by recursively packaging set of results below constructor symbols. That insight has been realized at the technical level by introducing s-cterms as suitable semantic values, and giving a proof calculus able to derive reachable s-cterms from a given expression. Previous to full abstraction, we have proved a bunch of good properties of the semantics: polarity, compositionality, closedness under substitutions, correctness and completeness with respect to rewriting.

We expect our semantics to be a useful tool for CS-based program manipulation. We remark that, for instance, to justify the correctness of a CS-transformation by proving preservation of reachable values could be incorrect if transformations are to be used locally. Our semantics could be a better option, and we plan to explore this.

There are other aspects not yet accomplished that can be subject of future work. In the paper, 'compositionality' refers to expressions wrt its subexpressions, and not to programs obtained by joining others, an interesting topic related to modularity (see e.g. [2]). We plan also to extend our approach to consider semantics and notions of observations that give a more active role to variables (as happens in [2, 1, 11]) taking into account that, for instance, in narrowing-based operational procedures, variables are subject of narrowing substitutions. Dropping the constructor restriction is also interesting, replacing the role of constructor values by appropriate alternatives. Finally, incorporating s-expressions to the syntax of programs, as discussed in Sect. 3.2, could lead to more expressive programs.

References

1. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Proc. LOPSTR'02*, p. 1–16, Springer LNCS 2664, 2003.
2. M. Alpuente, M. Falaschi, M. Ramis, and G. Vidal. Narrowing approximations as an optimization for equational logic programs. In *Proc. PLILP'93*, p. 391–409. Springer LNCS 714, 1993.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. ALP'97*, p. 16–30. Springer LNCS 1298, 1997.
4. S. Antoy, P. J. Iranzo, and B. Massey. Improving the efficiency of non-deterministic computations. *ENTCS, 64*, 2002.
5. G. Boudol. Une semantique pour les arbres non deterministes. In *Proc. CAAP'81*, p. 147–161, Springer LNCS 1981.
6. G. Boudol. Computational semantics of term rewriting systems. In *Algebraic methods in semantics*, p. 169–236, Camb. Univ. Press, 1986.
7. B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS*, p. 122–138, 2007.
8. M. Clavel et al. (eds). *All About Maude* , Springer LNCS 4350, 2007.
9. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. of Logic Programming*, 40(1):47–87, 1999.
10. M. Hanus. Multi-paradigm declarative languages. In *Proc.ICLP 2007*, p. 45–75. Springer LNCS 4670, 2007.
11. M. Hanus and S. Lucas. An evaluation semantics for narrowing-based functional logic languages. *J. Funct. and Logic Prog.*, 2001(2), 2001.
12. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs.* Birkhäuser Verlag, 1993.
13. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Bundles pack tighter than lists. In *Draft Proc. TFP'07*, 2007.
14. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.
15. S.-O. Nyström. There is no fully abstract fixpoint semantics for non-deterministic languages with infinite computations. *Inf. Process. Lett.*, 60(6):289–293, 1996.
16. E. Ohlebusch. *Advanced topics in term rewriting.* Springer-Verlag, 2002.
17. G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):225–255, 1977.
18. J. Reynolds. *Theories of Programing Languages.* Camb. Univ. Press, 1998.

A Proofs of the results

During this section we will use the symbol $\equiv$ to refer to syntactic equality between two elements, that is, occurence of the same symbols in the same positions.

Lemma 6. *For any $st \in SCTerm$ we have that $st \twoheadrightarrow st$.*

Proof. We proceed by induction on the structure of st. If $st \equiv \emptyset$ then $st \equiv \emptyset \twoheadrightarrow \emptyset \equiv st$, by E. Otherwise $st \equiv \{est_1, \ldots, est_n\}$ and we have two possibilities:

a) $n = 1$: Then if $st \equiv \{X\}$ for some $X \in \mathcal{V}$ we are done as $\{X\} \twoheadrightarrow \{X\}$ by RR. Otherwise $st \equiv \{c(st_1, \ldots, st_m)\}$ and we can apply the IH to each of them to get $st_i \twoheadrightarrow st_i$ and prove $\{c(st_1, \ldots, st_m)\} \twoheadrightarrow \{c(st_1, \ldots, st_m)\}$ by DC.
b) $n > 1$: Then each est_i must have one of the following shapes:
   i) $est_i \equiv X$ for some $X \in \mathcal{V}$: Then $\{est_i\} \equiv \{X\} \twoheadrightarrow \{X\} \equiv \{est_i\}$ by RR.
   ii) $est_i \equiv c(st'_1, \ldots, st'_m)$ and we can apply the IH to each of them to get $st'_i \twoheadrightarrow st'_i$ and prove $\{est_i\} \equiv \{c(st'_1, \ldots, st'_m)\} \twoheadrightarrow \{c(st'_1, \ldots, st'_m)\} \equiv \{est_i\}$ by DC.

   Therefore we have proved $\{est_i\} \twoheadrightarrow \{est_i\}$ for each est_i, hence we can apply Less to get $st \equiv \{est_1, \ldots, est_n\} \twoheadrightarrow \{est_1\} \cup \ldots \cup \{est_n\} \equiv \{est_1, \ldots, est_n\} \equiv st$.

Lemma 7 (Basic properties of $\sqsubseteq$).

- *If $\forall i \in \{1, \ldots, n\} \exists j \in \{1, \ldots, m\}$ such that $se_i \sqsubseteq se'_j$ then $se_1 \cup \ldots \cup se_n \sqsubseteq se'_1 \cup \ldots \cup se'_m$.*
- *If $se \sqsubseteq se'$ and $se' \subseteq se''$ then $se \sqsubseteq se''$.*
- *If $se \subseteq se'$ and $se' \sqsubseteq se''$ then $se \sqsubseteq se''$. As a result $se \subseteq se'$ implies $se \sqsubseteq se'$.*

Proof. By definition of $\sqsubseteq$ and basic set reasoning.

Lemma 8. *Under any program $\mathcal{P}$, $\forall st, st' \in SCTerm$ $st' \sqsubseteq st$ iff $st \twoheadrightarrow st'$.*

Proof. Assume $st' \sqsubseteq st$, then by Lemma 6 we have $st \twoheadrightarrow st$, hence $st \twoheadrightarrow st'$ by Prop. 1.
Concerning the other implication, assume $st \twoheadrightarrow st'$, we proceed by induction on the structure of $st \twoheadrightarrow st'$. The base cases for E, RR and DC are trivial. In the inductive case for DC we have $st \equiv \{c(st_1, \ldots, st_n)\} \twoheadrightarrow \{c(st'_1, \ldots, st'_n)\} \equiv st'$, and we may apply the IH to each $st_i \twoheadrightarrow st'_i$ to get $st'_i \sqsubseteq st_i$, hence $st \sqsubseteq st'$ by definition of $\sqsubseteq$. In the case for More we have $st \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st'$ and we can apply the IH to each $st \twoheadrightarrow st_i$ to get $st_i \sqsubseteq st$, hence $st' \equiv st_1 \cup \ldots \cup st_n \sqsubseteq st$ by Lemma 7. Finally, in the case for Less we have $st \twoheadrightarrow st_1 \cup \ldots \cup st_m \equiv st'$ using $\{esa_1, \ldots, esa_m\} \subseteq st$. Then we can apply the IH to each $\{esa_i\} \twoheadrightarrow st_i$ to get $st_i \sqsubseteq \{esa_i\}$, hence $st_1 \cup \ldots \cup st_m \sqsubseteq \{esa_1\} \cup \ldots \{esa_m\} \equiv \{esa_1, \ldots, esa_m\} \subseteq st$ by Lemma 7, and so $st' \equiv st_1 \cup \ldots \cup st_m \sqsubseteq st$ by Lemma 7 again.

A.1 For Subsection 3.1

Given a set se we use the notation $\sharp se$ to express its cardinality.

We use the depth of an expression that is the number of symbols of the signature contained in such expresion. Formally:

Definition 3 (Depth of expresions). *The depth of an expression e is defined as:*

- *$depth(\bot) = depth(X) = depth(h) = 0$, for $X \in \mathcal{V}$, $h \in \Sigma^0$*
- *$depth(h(e_1, \ldots, e_n)) = 1 + \sum_{i=1}^{n} depth(e_i)$, for $h \in \Sigma^n$*

Proposition 8. *For any $e \in Exp$ it holds $\tilde{e} \in SExp$ and $var(\tilde{e}) = var(e)$.*

Proof. These properties are quit clear by construction. A formal proof can be done by induction on the depth of the expression e. The base cases are $\bot$, $X \in \mathcal{V}$ and $h \in \Sigma^0$ for which both properties clearly hold. For the case $e = h(e_1, \ldots, e_n)$ with $n > 0$ we have $\tilde{e} = \{h(\tilde{e}_1, \ldots, \tilde{e}_n)\}$ and we can apply IH to each e_i.

Proposition 9. *For any $e \in Exp$ it holds $flat(\tilde{e}) = \{e\}$.*

Proof. Again we proceed by induction on the depth of the expression e. There are three base cases and we only have to apply sequentially the definitions of $\tilde{}$ and $flat$:

- $e = \bot$: we have $flat(\tilde{\bot}) = flat(\emptyset) = \{\bot\}$.
- $e = X \in \mathcal{V}$: we have $flat(\tilde{X}) = flat(\{X\}) = \{X\}$.
- $e = h \in \Sigma^0$: we have $flat(\tilde{h}) = flat(\{h\}) = \{h\}$.

For the a depth greater than 0 consider $e = h(e_1, \ldots, e_n)$. We have the following chain:

$$\begin{array}{lll} flat(\widetilde{h(e_1, \ldots, e_n)}) = flat(\{h(\tilde{e}_1, \ldots, \tilde{e}_n)\}) & & \text{(def. of } \tilde{\ }\text{)} \\ = flat(h(\tilde{e}_i, \ldots, \tilde{e}_n)) & & \text{(def. of } flat \text{ for } eSExp\text{)} \\ = \{h(e'_1, \ldots, e'_n) \mid e'_i \in flat(\tilde{e}_i)\} & & \text{(def. of } flat \text{ for } SExp\text{)} \\ = \{h(e'_1, \ldots, e'_n) \mid e'_i \in \{e_i\}\} & & \text{(IH)} \\ = \{h(e_1, \ldots, e_n)\} & & \text{sets manipulation} \end{array}$$

Proposition 10. *Given $\sigma \in SSubst$, $se \in SExp$ and $ese \in eSExp$ we have both $se\sigma \in SExp$ and $ese\sigma \in SExp$.*

Proof. It is a direct application of the definition of application of substitutions.

Lemma 9. $\forall e \in Exp_\perp, \sigma \in Subst_\perp, \mathcal{C} \in Cntxt$:

i) $\widetilde{e\sigma} \equiv \tilde{e}\tilde{\sigma}$.
ii) $\widetilde{\mathcal{C}[e]} \equiv \tilde{\mathcal{C}}[\tilde{e}]$.

Proof.

i) We proceed by induction on the structure of e. Concerning the base cases:
   - $e \equiv \perp$: Then $\widetilde{e\sigma} \equiv \widetilde{\perp\sigma} \equiv \tilde{\perp} \equiv \emptyset \equiv \emptyset\tilde{\sigma} \equiv \tilde{\perp}\tilde{\sigma} \equiv \tilde{e}\tilde{\sigma}$
   - $e \equiv X \in \mathcal{V}$: Then $\widetilde{e\sigma} \equiv \widetilde{X\sigma} \equiv \widetilde{\sigma(X)} \equiv \tilde{\sigma}(X) \equiv X\tilde{\sigma} \equiv \{X\}\tilde{\sigma} \equiv \tilde{X}\tilde{\sigma} \equiv \tilde{e}\tilde{\sigma}$
   - $e \equiv h$: Then $\widetilde{e\sigma} \equiv \widetilde{h\sigma} \equiv \tilde{h} \equiv \{h\} \equiv \{h\}\tilde{\sigma} \equiv \tilde{e}\tilde{\sigma}$

   Concening the inductive step, this happens when $e \equiv h(e_1, \ldots, e_n)$, then

$$\begin{array}{l} \widetilde{e\sigma} \equiv \widetilde{h(e_1, \ldots, e_n)\sigma} \equiv \widetilde{h(e_1\sigma, \ldots, e_n\sigma)} \equiv \{h(\widetilde{e_1\sigma}, \ldots, \widetilde{e_n\sigma})\} \\ \equiv_{IH} \{h(\tilde{e_1}\tilde{\sigma}, \ldots, \tilde{e_n}\tilde{\sigma})\} \equiv \{h(\tilde{e_1}, \ldots, \tilde{e_n})\}\tilde{\sigma} \equiv \widetilde{h(e_1, \ldots, e_n)}\tilde{\sigma} \equiv \tilde{e}\tilde{\sigma} \end{array}$$

ii) We proceed by induction on the structure of $\mathcal{C}$. The base case happens when $\mathcal{C} \equiv []$, then $\widetilde{\mathcal{C}[e]} \equiv \widetilde{[e]} \equiv \tilde{e} \equiv []\tilde{[e]} \equiv \tilde{[]}[\tilde{e}] \equiv \tilde{\mathcal{C}}[\tilde{e}]$. Concerning the inductive step, this happens when $\mathcal{C} \equiv h(e_1, \ldots, \mathcal{C}', \ldots, e_n)$, then

$$\begin{array}{ll} \widetilde{\mathcal{C}[e]} \equiv h(e_1, \ldots, \widetilde{\mathcal{C}'[e]}, \ldots, e_n) \equiv \{h(\widetilde{e_1}, \ldots, \widetilde{\mathcal{C}'[e]}, \ldots, \widetilde{e_n})\} & \\ \equiv \{h(\widetilde{e_1}, \ldots, \tilde{\mathcal{C}}'[\tilde{e}], \ldots, \widetilde{e_n})\} & \text{by IH} \\ \equiv \{h(\widetilde{e_1}, \ldots, \tilde{\mathcal{C}}', \ldots, \widetilde{e_n})\}[\tilde{e}] \equiv \tilde{\mathcal{C}}[\tilde{e}] & \end{array}$$

A.2 For Subsection 3.2

Proof (For Proposition 1 (Polarity)). We proceed by induction on the structure of $se \twoheadrightarrow st$. Concerning the base cases:

E Then we have $se \twoheadrightarrow \emptyset \equiv st$, and so $st' \sqsubseteq st$ implies $st' \equiv \emptyset$ too, therefore $se \twoheadrightarrow \emptyset \equiv st'$ by E again.
RR Then we have $se \equiv \{X\} \twoheadrightarrow \{X\} \equiv st$, and so $st' \sqsubseteq st$ implies $st' \equiv \emptyset$ or $st' \equiv \{X\}$. Besides $se \equiv \{X\} \sqsubseteq se'$ implies $X \in se'$ and so $\sharp se' \geq 1$. Let us consider the different possibilities:
   a) $st' \equiv \emptyset$: Then $se \twoheadrightarrow \emptyset \equiv st'$ by E.
   b) $st' \equiv \{X\}$:
      i) $\sharp se' = 1$: Then $X \in se'$ implies $se' \equiv \{X\}$ and so $se' \equiv \{X\} \twoheadrightarrow \{X\} \equiv st'$ by RR.
      ii) $\sharp se' > 1$: Then we can use $X \in se'$ to do

$$\dfrac{\dfrac{}{\{X\} \twoheadrightarrow \{X\}}\ \text{RR}}{se' \twoheadrightarrow \{X\} \equiv st'}\ \textsc{Less}$$

DC Similar to the previous case just changing X for c and RR for DC.

Concerning the inductive steps:

DC If $st' \equiv \emptyset$ then we proceed like in the case for E. Otherwise, we have

$$\frac{se_1 \twoheadrightarrow st_1 \quad \dots \quad se_n \twoheadrightarrow st_n}{se \equiv \{c(se_1, \dots, se_n)\} \twoheadrightarrow \{c(st_1, \dots, st_n)\} \equiv st} \text{ DC}$$

Then $se \sqsubseteq se'$ implies that $\exists c(se'_1, \dots, se'_n) \in se'$ such that $\forall i \in \{1, \dots, n\}, se_i \sqsubseteq se'_i$, and so $\sharp se' \geq 1$. Besides $st' \sqsubseteq st$ implies $\forall est'_j \in st', est'_j \equiv c(st'_{1j}, \dots, st'_{nj})$ such that $\forall i \in \{1, \dots, n\}, st'_{ij} \sqsubseteq st_i$. But then, if $st' \equiv \{est'_1, \dots, est'_m\}$, we can apply the IH $\forall j \in \{1, \dots, m\}, \forall i \in \{1, \dots, n\}$ over $se_i \twoheadrightarrow st_i$ with $se_i \sqsubseteq se'_i$ and $st'_{ij} \sqsubseteq st_i$ to get $se'_i \twoheadrightarrow st'_{ij}$. And we have the following possibilities:

a) $\sharp se' = 1$: Then $c(se'_1, \dots, se'_n) \in se'$ implies $se' \equiv \{c(se'_1, \dots, se'_n)\}$.

i) $\sharp st' = m = 1$: Then we can do

$$\frac{\dfrac{IH}{se'_1 \twoheadrightarrow st'_{11}} \quad \dots \quad \dfrac{IH}{se'_n \twoheadrightarrow st'_{n1}}}{se' \equiv \{c(se'_1, \dots, se'_n)\} \twoheadrightarrow \{c(st'_{11}, \dots, st'_{n1})\} \equiv st'} \text{ DC}$$

ii) $\sharp st' = m > 1$: Then we can do

$$\frac{\dfrac{\dfrac{IH}{se'_1 \twoheadrightarrow st'_{11}} \quad \dots \quad \dfrac{IH}{se'_n \twoheadrightarrow st'_{n1}}}{\{c(se'_1, \dots, se'_n)\} \twoheadrightarrow \{c(st'_{11}, \dots, st'_{n1})\}} \text{ DC} \quad \dots \quad \dfrac{\dfrac{IH}{se'_1 \twoheadrightarrow st'_{1m}} \quad \dots \quad \dfrac{IH}{se'_n \twoheadrightarrow st'_{nm}}}{\{c(se'_1, \dots, se'_n)\} \twoheadrightarrow \{c(st'_{1m}, \dots, st'_{nm})\}} \text{ DC}}{se' \equiv \{c(se'_1, \dots, se'_n)\} \twoheadrightarrow \{c(st'_{11}, \dots, st'_{n1})\} \cup \dots \cup \{c(st'_{1m}, \dots, st'_{nm})\} \equiv st'} \text{ More}$$

b) $\sharp se' > 1$: Then we can prove $\{c(se'_1, \dots, se'_n)\} \twoheadrightarrow st'$ like in case *a)*, so as $c(se'_1, \dots, se'_n) \in se'$ we can apply Less to get $se' \twoheadrightarrow st'$.

More If $st' \equiv \emptyset$ then we proceed like in the case for E. Otherwise $st' \equiv \{est'_1, \dots, est'_m\}$ with $m > 0$ and we have

$$\frac{se \twoheadrightarrow st_1 \quad \dots \quad se \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \dots \cup st_n \equiv st} \text{ More}$$

Then $st' \sqsubseteq st$ implies $\forall est'_j \in st', \exists est_j \in st \equiv st_1 \cup \dots \cup st_n$ such that $est'_j \sqsubseteq est_j$, therefore $\exists i \in \{1, \dots, n\}$ such that $est'_j \sqsubseteq est_j \in st_i$, hence $\{est'_j\} \sqsubseteq st_i$. But then we can apply the IH over $se \twoheadrightarrow st_i$ with $\{est'_j\} \sqsubseteq st_i$ and $se \sqsubseteq se'$ to get that $se' \twoheadrightarrow \{est'_j\}$. We can do it for every $est'_j \in st'$, and build the following proof

$$\frac{\dfrac{IH}{se' \twoheadrightarrow \{est'_1\}} \quad \dots \quad \dfrac{IH}{se' \twoheadrightarrow \{est'_m\}}}{se' \twoheadrightarrow \{est'_1\} \cup \dots \cup \{est'_m\} \equiv st'} \text{ More}$$

Less If $st' \equiv \emptyset$ then we proceed like in the case for E. Otherwise $st' \equiv \{est'_1, \dots, est'_m\}$ with $m > 0$ and we have

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \quad \dots \quad \{esa_n\} \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \dots \cup st_n \equiv st} \text{ Less}$$

Then we can reason like in the case for More to get that $st' \sqsubseteq st$ implies that $\forall est'_j \in st', \exists i \in \{1, \dots, n\}$ such that $\{est'_j\} \sqsubseteq st_i$. Besides by Lemma 7 we have that $\forall i \in \{1, \dots, n\}, \{esa_i\} \subseteq \{esa_1, \dots, esa_n\} \subseteq se \sqsubseteq se'$ implies $\{esa_i\} \sqsubseteq se'$, and so we can apply the IH to each $\{esa_i\} \twoheadrightarrow st_i$ with $\{est'_j\} \sqsubseteq st_i$ and $\{esa_i\} \sqsubseteq se'$ to get $se' \twoheadrightarrow \{est'_j\}$, and build the following proof

$$\frac{\dfrac{IH}{se' \twoheadrightarrow \{est'_1\}} \quad \dots \quad \dfrac{IH}{se' \twoheadrightarrow \{est'_m\}}}{se' \twoheadrightarrow \{est'_1\} \cup \dots \cup \{est'_m\} \equiv st'} \text{ More}$$

ROR If $st' \equiv \emptyset$ then we proceed like in the case for E. Otherwise $st' \equiv \{est'_1, \dots, est'_m\}$ with $m > 0$ and we have

$$\frac{se_1 \twoheadrightarrow \tilde{p_1}\theta \quad \dots \quad se_n \twoheadrightarrow \tilde{p_n}\theta \quad \tilde{r}\theta \twoheadrightarrow st}{se \equiv \{f(se_1, \dots, se_n)\} \twoheadrightarrow st} \text{ ROR}$$

Then $se \sqsubseteq se'$ implies that $\exists f(se'_1, \dots, se'_n) \in se'$ such that $\forall i \in \{1, \dots, n\}, se_i \sqsubseteq se'_i$, and so $\sharp se' \geq 1$. But then by IH over each $se_i \twoheadrightarrow \tilde{p_i}\theta$ with $\tilde{p_i}\theta \sqsubseteq \tilde{p_i}\theta$ and $se_i \sqsubseteq se'_i$ we get $se'_i \twoheadrightarrow \tilde{p_i}\theta$. We can also apply the IH to $\tilde{r}\theta \twoheadrightarrow st$ with $st' \sqsubseteq st$ and $\tilde{r}\theta \sqsubseteq \tilde{r}\theta$, to get $\tilde{r}\theta \twoheadrightarrow st'$. Then we have the following possibilities:

a) $\sharp se' = 1$: Then $f(se'_1, \ldots, se'_n) \in se'$ implies $se' \equiv \{f(se'_1, \ldots, se'_n)\}$, and we can do

$$\frac{\frac{IH}{se'_1 \twoheadrightarrow \tilde{p_1}\theta} \quad \ldots \quad \frac{IH}{se'_n \twoheadrightarrow \tilde{p_n}\theta} \quad \frac{IH}{\tilde{r}\theta \twoheadrightarrow st'}}{se' \equiv \{f(se'_1, \ldots, se'_n)\} \twoheadrightarrow st'} \text{ ROR}$$

b) $\sharp se' > 1$: Then we can prove $\{f(se'_1, \ldots, se'_n)\} \twoheadrightarrow st'$ like in case *a)*, so as $f(se'_1, \ldots, se'_n) \in se'$ we can apply Less to get $se' \twoheadrightarrow st'$.

Lemma 10. *Under any program and* $\forall se \in SExp, ese \in ESExp, \sigma \in SSubst$*,if* $se\sigma \equiv \{ese\}$ *then* $\forall ese' \in se, ese'\sigma \equiv \{ese\} \equiv se\sigma$*.*

Proof. $\{ese\} \equiv \bigcup_{ese' \in se} ese'\sigma$, hence $\forall ese' \in se, ese'\sigma \equiv \{ese\} \equiv se\sigma$.

Lemma 11. *Under any program and* $\forall se \in SExp, st \in SCTerm, \sigma \in SSubst$*, if* $\exists ese \in se$ *such that* $ese\sigma \twoheadrightarrow st$ *then* $se\sigma \twoheadrightarrow st$*. In other words,* $\forall ese \in se, [\![ese\sigma]\!] \subseteq [\![se\sigma]\!]$*.*

Proof. If $ese \in se$ then $ese\sigma \subseteq \bigcup_{esa \in se} esa\sigma \equiv se\sigma$, hence $ese\sigma \sqsubseteq se\sigma$ by Lemma 7, and $[\![ese\sigma]\!] \subseteq [\![se\sigma]\!]$ by Prop. 1.

Proposition 11. *The relation* $\trianglelefteq$ *is a preorder but not a partial order under any CS.*

Proof. It is reflexive because obviously $\forall \sigma \in SSubst, \forall X \in \mathcal{V}, [\![\sigma(X)]\!] = [\![\sigma(X)]\!]$, it is transitive as a consecuence of the transitivity of $\subseteq$, but it is not antisymmetric because for example under $\{amb(X,Y) \to X, amb(X,Y) \to Y\}$ we have $[X/amb(a,b)] \trianglelefteq [X/amb(a, amb(a,b))]$ and $[X/amb(a, amb(a,b))] \trianglelefteq [X/amb(a,b)]$ but $[X/amb(a,b)] \neq [X/amb(a, amb(a,b))]$.

Proof (For Proposition 2 (Monotonicity of substitutions)). First of all, if $\sigma \sqsubseteq \sigma'$ then $\sigma \trianglelefteq \sigma'$, because if $\sigma \sqsubseteq \sigma'$ then $\forall X \in \mathcal{V}$ if $\sigma(X) \twoheadrightarrow st$ then as $\sigma(X) \sqsubseteq \sigma'(X)$ then we can apply Prop. 1 to get $\sigma'(X) \twoheadrightarrow st$.

All that is left is proving that $\sigma \trianglelefteq \sigma'$ implies that forall $st \in SCTerm$ such that $se\sigma \twoheadrightarrow st$ then $se\sigma' \twoheadrightarrow st$. We proceed by induction on the structure of $se\sigma \twoheadrightarrow st$, the base cases are the following:

E Then $st \equiv \emptyset$, hence $se\sigma' \twoheadrightarrow \emptyset \equiv st$ by E.

RR Then $se\sigma \equiv \{X\} \twoheadrightarrow \{X\} \equiv st$, and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{X\}$, hence $\forall ese \in se, ese \in \mathcal{V}$. Besides $se\sigma \equiv \{X\}$ implies $se \not\equiv \emptyset$, therefore $\exists ese \in se, ese \equiv Y \in \mathcal{V} \wedge Y\sigma \equiv ese\sigma \equiv \{X\}$, and so $\sigma(Y) \equiv Y\sigma \equiv \{X\} \twoheadrightarrow st$ by hypothesis. But then $\sigma \trianglelefteq \sigma'$ implies $st \in [\![\sigma(Y)]\!] \subseteq [\![\sigma'(Y)]\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

DC Then $se\sigma \equiv \{c\} \twoheadrightarrow \{c\} \equiv st$, and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{c\}$. We have two possibilities:

a) $se \cap \mathcal{V} \neq \emptyset$: Then given some $Y \in se \cap \mathcal{V}$ we have $\sigma(Y) \equiv Y\sigma \equiv \{c\} \twoheadrightarrow st$ by hyphotesis. But then $\sigma \trianglelefteq \sigma'$ implies $st \in [\![\sigma(Y)]\!] \subseteq [\![\sigma'(Y)]\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

b) $se \cap \mathcal{V} = \emptyset$: Then $\forall ese \in se, ese\sigma \equiv \{c\}$ implies $\forall ese \in se, ese \equiv \{c\}$. Besides $se\sigma \equiv \{c\}$ implies $se \not\equiv \emptyset$, therefore $\exists ese \in se, ese\sigma' \equiv \{c\}\sigma' \equiv \{c\} \twoheadrightarrow st$ by hypothesis , but then $st \in [\![ese\sigma']\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

Concerning the inductive steps:

DC Then we have

$$\frac{se_1 \twoheadrightarrow st_1 \quad \ldots \quad se_n \twoheadrightarrow st_n}{se\sigma \equiv \{c(se_1, \ldots, se_n)\} \twoheadrightarrow \{c(st_1, \ldots, st_n)\} \equiv st} \text{ DC}$$

and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{c(se_1, \ldots, se_n)\}$. We have two possibilities:

a) $se \cap \mathcal{V} \neq \emptyset$: Then given some $Y \in se \cap \mathcal{V}$ we have $\sigma(Y) \equiv Y\sigma \equiv \{c(se_1, \ldots, se_n)\} \twoheadrightarrow st$ by hypothesis. But then $\sigma \trianglelefteq \sigma'$ implies $st \in [\![\sigma(Y)]\!] \subseteq [\![\sigma'(Y)]\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

b) $se \cap \mathcal{V} = \emptyset$: Then $se\sigma \equiv \{c(se_1, \ldots, se_n)\}$ implies $se \not\equiv \emptyset$. Therefore $\exists ese \in se = (se \setminus \mathcal{V})$ such that $ese\sigma \equiv \{c(se_1, \ldots, se_n)\}$, which implies $ese \equiv c(se'_1, \ldots, se'_n)$ such that $\forall i, se'_i\sigma \equiv se_i \twoheadrightarrow st_i$, to which we can apply the IH to get $se'_i\sigma' \twoheadrightarrow st_i$, and build the following proof:

$$\frac{se'_1\sigma' \twoheadrightarrow st_1 \quad \ldots \quad se'_n\sigma' \twoheadrightarrow st_n}{ese\sigma' \equiv \{c(se'_1\sigma', \ldots, se'_n\sigma')\} \twoheadrightarrow \{c(st_1, \ldots, st_n)\} \equiv st} \text{ DC}$$

But then $st \in [\![ese\sigma']\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

More Then we have

$$\frac{se\sigma \twoheadrightarrow st_1 \quad \dots \quad se\sigma \twoheadrightarrow st_n}{se\sigma \twoheadrightarrow st_1 \cup \dots \cup st_n \equiv st} \text{ More}$$

But then we can apply the IH to each $se\sigma \twoheadrightarrow st_i$ to get $se\sigma' \twoheadrightarrow st_i$, and build the following proof:

$$\frac{se\sigma' \twoheadrightarrow st_1 \quad \dots \quad se\sigma' \twoheadrightarrow st_n}{se\sigma' \twoheadrightarrow st_1 \cup \dots \cup st_n \equiv st} \text{ More}$$

Less Then we have

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \quad \dots \quad \{esa_n\} \twoheadrightarrow st_m}{se\sigma \twoheadrightarrow st_1 \cup \dots \cup st_m \equiv st} \text{ Less}$$

for some $\{esa_1, \dots, esa_m\} \subseteq se\sigma$ and $\sharp se\sigma \geq 2$. Then for any $esa_i \in \{esa_1, \dots, esa_m\} \subseteq se\sigma$ we have two possibilities:

a) $esa_i \in \sigma(X)$ for some $X \in se$: But then

$$\begin{array}{ll} st_i \in [\![\{esa_i\}]\!] \subseteq [\![\sigma(X)]\!] & \text{by Lemma 7 and Prop. 1, as } \{esa_i\} \subseteq \sigma(X) \\ \subseteq [\![\sigma'(X)]\!] = [\![X\sigma']\!] & \text{as } \sigma \trianglelefteq \sigma' \\ \subseteq [\![se\sigma']\!] & \text{by Lemma 11, as } X \in se \end{array}$$

b) $esa_i \equiv h(se_1\sigma, \dots, se_n\sigma)$ for some $h(se_1, \dots, se_n) \in se$. But then $\{h(se_1, \dots, se_n)\}\sigma \equiv \{h(se_1\sigma, \dots, se_n\sigma)\} \equiv \{esa_i\} \twoheadrightarrow st_i$ by hypothesis, and we can apply the IH to get $\{h(se_1, \dots, se_n)\}\sigma' \twoheadrightarrow st_i$. But then $h(se_1, \dots, se_n) \in se$ implies $st_i \in [\![\{h(se_1, \dots, se_n)\}\sigma']\!] = [\![h(se_1, \dots, se_n)\sigma']\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

Therefore $\forall i \in \{1, \dots, m\}, se\sigma' \twoheadrightarrow st_i$, and we can build the following proof:

$$\frac{se\sigma' \twoheadrightarrow st_1 \quad \dots \quad se\sigma' \twoheadrightarrow st_m}{se\sigma' \twoheadrightarrow st_1 \cup \dots \cup st_m \equiv st} \text{ More}$$

ROR Then we have

$$\frac{se_1 \twoheadrightarrow \tilde{p_1}\theta \quad \dots \quad se_n \twoheadrightarrow \tilde{p_n}\theta \quad \tilde{r}\theta \twoheadrightarrow st}{se\sigma \equiv \{f(se_1, \dots, se_n)\} \twoheadrightarrow st} \text{ ROR}$$

and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{f(se_1, \dots, se_n)\}$. We have two possibilities:

a) $se \cap \mathcal{V} \neq \emptyset$: Then given some $Y \in se \cap \mathcal{V}$ we have $\sigma(Y) \equiv Y\sigma \equiv \{f(se_1, \dots, se_n)\} \twoheadrightarrow st$ by hypothesis. But then $\sigma \trianglelefteq \sigma'$ implies $st \in [\![\sigma(Y)]\!] \subseteq [\![\sigma'(Y)]\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

b) $se \cap \mathcal{V} = \emptyset$: Then $se\sigma \equiv \{f(se_1, \dots, se_n)\}$ implies $se \not\equiv \emptyset$. Therefore $\exists ese \in se = (se \setminus \mathcal{V})$ such that $ese\sigma \equiv \{f(se_1, \dots, se_n)\}$, which implies $ese \equiv f(se'_1, \dots, se'_n)$ such that $\forall i, se'_i\sigma \equiv se_i \twoheadrightarrow \tilde{p_i}\theta$, to which we can apply the IH to get $se'_i\sigma' \twoheadrightarrow \tilde{p_i}\theta$, and build the following proof:

$$\frac{se'_1\sigma' \twoheadrightarrow \tilde{p_1}\theta \quad \dots \quad se'_n\sigma' \twoheadrightarrow \tilde{p_n}\theta \quad \dfrac{hypothesis}{\tilde{r}\theta \twoheadrightarrow st}}{ese\sigma' \equiv \{f(se'_1\sigma', \dots, se'_n\sigma')\} \twoheadrightarrow st} \text{ ROR}$$

But then $st \in [\![ese\sigma']\!] \subseteq [\![se\sigma']\!]$ by Lemma 11.

Lemma 12. *For any $\sigma \in SSubst, \theta \in [\![\sigma]\!]$ we have $\theta \trianglelefteq \sigma$.*

Proof. It is enough to check that $\forall X \in \mathcal{V}$, $[\![\theta(X)]\!] \subseteq [\![\sigma(X)]\!]$. That happens because given any $st \in SCTerm$ such that $\theta(X) \twoheadrightarrow st$, as $\theta \in [\![\sigma]\!]$ implies $\theta \in SCSubst$ which implies $\theta(X) \in SCTerm$, then $st \sqsubseteq \theta(X)$ by Lemma 8. But $\theta \in [\![\sigma]\!]$ implies $\sigma(X) \twoheadrightarrow \theta(X)$, hence we can apply Prop. 1 to get $\sigma(X) \twoheadrightarrow st$.

The following lemma will be used to prove compositionality.

Lemma 13. *$\forall se_1, se_2 \in SExp$ such that $se_1 \subseteq se_2$ we have that $\forall s\mathcal{C} \in sCntxt, [\![s\mathcal{C}[se_1]]\!] \subseteq [\![s\mathcal{C}[se_2]]\!]$.*

Proof. If $se_1 = se_2$ then $s\mathcal{C}[se_1] \equiv s\mathcal{C}[se_2]$ and so the result trivially holds.
On the other hand, for the case when $se_1 \subset se_2$, we will see that for any $s\mathcal{C} \in sCntxt$ and any $st \in SCTerm$ such that $s\mathcal{C}[se_1] \twoheadrightarrow st$ then $s\mathcal{C}[se_2] \twoheadrightarrow st$, by induction on the size K of $s\mathcal{C}[se_1] \twoheadrightarrow st$.

Base cases $K = 1$: Let us see which was the rule applied at the root of the proof for $s\mathcal{C}[se_1] \twoheadrightarrow st$.

E : Then $st \equiv \emptyset$ but then $s\mathcal{C}[se_2] \twoheadrightarrow \emptyset \equiv st$ by E.

RR : Then $s\mathcal{C}[se_1] \equiv \{X\}$ and so $s\mathcal{C} = []$ and $s\mathcal{C}[se_1] \equiv se_1 \equiv \{X\}$. Hence $\sharp se_1 = 1$, which combined with $se_1 \subseteq se_2$ implies $\sharp se_1 \geq 2$, and so

$$\dfrac{\dfrac{}{hypothesis}}{\dfrac{\{X\} \equiv se_1 \equiv s\mathcal{C}[se_1] \twoheadrightarrow st}{s\mathcal{C}[se_2] \equiv se_2 \twoheadrightarrow \{X\} \equiv st}\ \textsc{Less}}$$

as $\{X\} \equiv se_1 \subset se_2$.

DC : Then $s\mathcal{C}[se_1] \equiv \{c\}$ and so $s\mathcal{C} = []$, and the hypothesis was

$$\dfrac{}{s\mathcal{C}[se_1] \equiv se_1 \equiv \{c\} \twoheadrightarrow \{c\} \equiv st}\ \text{DC}$$

Hence $\sharp se_1 = 1$, which combined with $se_1 \subseteq se_2$ implies $\sharp se_1 \geq 2$, and so

$$\dfrac{\dfrac{}{hypothesis}}{\dfrac{\{c\} \equiv se_1 \equiv s\mathcal{C}[se_1] \twoheadrightarrow st}{s\mathcal{C}[se_2] \equiv se_2 \twoheadrightarrow \{c\} \equiv st}\ \textsc{Less}}$$

as $\{X\} \equiv se_1 \subset se_2$.

Inductive step $K > 1$: Let us see which was the rule applied at the root of the proof for $s\mathcal{C}[se_1] \twoheadrightarrow st$.

DC If $s\mathcal{C} = []$ then $s\mathcal{C}[se_1] \equiv se_1 \equiv \{c(se'_1, \ldots, se'_n)\}$ and so $\sharp se_1 = 1$, which combined with $se_1 \subseteq se_2$ implies $\sharp se_1 \geq 2$, and so

$$\dfrac{\dfrac{}{hypothesis}}{\dfrac{\{c(se'_1, \ldots, se'_n)\} \equiv se_1 \equiv s\mathcal{C}[se_1] \twoheadrightarrow st}{s\mathcal{C}[se_2] \equiv se_2 \twoheadrightarrow st}\ \textsc{Less}}$$

as $\{c(se'_1, \ldots, se'_n)\} \equiv se_1 \subseteq se_2$. Otherwise if $s\mathcal{C} \neq []$ then the hypothesis was

$$\dfrac{se'_1 \twoheadrightarrow st'_1 \ \ldots\ s\mathcal{C}'[se_1] \twoheadrightarrow st' \ \ldots\ se'_n \twoheadrightarrow st'_n}{s\mathcal{C}[se_1] \equiv \{c(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_n)\} \twoheadrightarrow c(st'_1, \ldots, st', \ldots, st'_n)\}}\ \text{DC}$$

but then we can apply the IH to $s\mathcal{C}'[se_1] \twoheadrightarrow st'$ to get $s\mathcal{C}'[se_2] \twoheadrightarrow st'$, so we can build

$$\dfrac{\dfrac{}{hypothesis}\ \ldots\ \dfrac{}{IH}\ \ldots\ \dfrac{}{hypothesis}}{s\mathcal{C}[se_2] \equiv \{c(se'_1, \ldots, s\mathcal{C}'[se_2], \ldots, se'_n)\} \twoheadrightarrow c(st'_1, \ldots, st', \ldots, st'_n)\}}\ \text{DC}$$

More Then we have

$$\dfrac{s\mathcal{C}[se_1] \twoheadrightarrow st_1 \ \ldots\ s\mathcal{C}[se_1] \twoheadrightarrow st_n}{s\mathcal{C}[se_1] \twoheadrightarrow st_1 \cup \ldots \cup st_n}\ \textsc{More}$$

but then we can apply the IH to each $s\mathcal{C}[se_1] \twoheadrightarrow st_i$ to get $s\mathcal{C}[se_2] \twoheadrightarrow st_i$ and build

$$\dfrac{\dfrac{}{IH}\ \ldots\ \dfrac{}{IH}}{s\mathcal{C}[se_2] \twoheadrightarrow st_1 \cup \ldots \cup st_n}\ \textsc{More}$$

Less If $s\mathcal{C} = []$ then the hypothesis was

$$\dfrac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots\ \{esa_m\} \twoheadrightarrow st_m}{s\mathcal{C}[se_1] \equiv se_1 \twoheadrightarrow st_1 \cup \ldots \cup st_m}\ \textsc{Less}$$

with $\{esa_1, \ldots, esa_m\} \subseteq se_1 \subset se_2$, but then

$$\dfrac{\dfrac{}{hypothesis}\ \ldots\ \dfrac{}{hypothesis}}{\dfrac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots\ \{esa_m\} \twoheadrightarrow st_m}{s\mathcal{C}[se_2] \equiv se_2 \twoheadrightarrow st_1 \cup \ldots \cup st_m}\ \textsc{Less}}$$

Otherwise if $s\mathcal{C} \neq []$ then the hypothesis was

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots \ \{esa_m\} \twoheadrightarrow st_m}{s\mathcal{C}[se_1] \equiv \{ese_1, \ldots, h(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_k), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m} \textsc{Less}$$

Now we have two possibilities. If $h(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_k) \notin \{esa_1, \ldots, esa_m\}$, then we can do

$$\frac{\dfrac{hypothesis}{\{esa_1\} \twoheadrightarrow st_1} \ \ldots \ \dfrac{hypothesis}{\{esa_m\} \twoheadrightarrow st_m}}{s\mathcal{C}[se_2] \equiv \{ese_1, \ldots, h(se'_1, \ldots, s\mathcal{C}'[se_2], \ldots, se'_k), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m} \textsc{Less}$$

because then neither $h(se'_1, \ldots, s\mathcal{C}'[se_2], \ldots, se'_k)$ nor $s\mathcal{C}'[se_2]$ in general are involved in the premises. Otherwise if $h(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_k) \in \{esa_1, \ldots, esa_m\}$ then consider the corresponding premise $h(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_k) \twoheadrightarrow st_j$. Then we can take $s\mathcal{C}'' = h(se'_1, \ldots, s\mathcal{C}', \ldots, se'_k)$ for which $s\mathcal{C}''[se_1] \twoheadrightarrow st_j$ has a proof of size less than K, and apply the IH to it to get $s\mathcal{C}''[se_2] \twoheadrightarrow st_j$, so we can build

$$\frac{\dfrac{hypothesis}{\{esa_1\} \twoheadrightarrow st_1} \ \ldots \ \dfrac{IH}{h(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_k) \equiv s\mathcal{C}''[se_2] \twoheadrightarrow st_j} \ \ldots \ \dfrac{hypothesis}{\{esa_m\} \twoheadrightarrow st_m}}{s\mathcal{C}[se_2] \equiv \{ese_1, \ldots, h(se'_1, \ldots, s\mathcal{C}'[se_2], \ldots, se'_k), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_j \cup \ldots \cup st_m} \textsc{Less}$$

ROR If $s\mathcal{C} = []$ then $s\mathcal{C}[se_1] \equiv se_1 \equiv \{f(se'_1, \ldots, se'_n)\}$ and so $\sharp se_1 = 1$, which combined with $se_1 \subseteq se_2$ implies $\sharp se_1 \geq 2$, and so

$$\frac{\dfrac{hypothesis}{\{f(se'_1, \ldots, se'_n)\} \equiv se_1 \equiv s\mathcal{C}[se_1] \twoheadrightarrow st}}{s\mathcal{C}[se_2] \equiv se_2 \twoheadrightarrow st} \textsc{Less}$$

as $\{f(se'_1, \ldots, se'_n)\} \equiv se_1 \subseteq se_2$. Otherwise if $s\mathcal{C} \neq []$ then the hypothesis was

$$\frac{se'_1 \twoheadrightarrow \tilde{p_1}\theta \ \ldots \ s\mathcal{C}'[se_1] \twoheadrightarrow \tilde{p'}\theta \ \ldots \ se'_n \twoheadrightarrow \tilde{p_n}\theta \ \tilde{r}\theta \twoheadrightarrow st}{s\mathcal{C}[se_1] \equiv \{f(se'_1, \ldots, s\mathcal{C}'[se_1], \ldots, se'_n)\} \twoheadrightarrow st} \text{ ROR}$$

but then we can apply the IH to $s\mathcal{C}'[se_1] \twoheadrightarrow \tilde{p'}\theta$ to get $s\mathcal{C}'[se_2] \twoheadrightarrow \tilde{p'}\theta$, so we can build

$$\frac{\dfrac{hypothesis}{se'_1 \twoheadrightarrow \tilde{p_1}\theta} \ \ldots \ \dfrac{IH}{s\mathcal{C}'[se_2] \twoheadrightarrow \tilde{p'}\theta} \ \ldots \ \dfrac{hypothesis}{se'_n \twoheadrightarrow \tilde{p_n}\theta} \ \dfrac{hypothesis}{\tilde{r}\theta \twoheadrightarrow st}}{s\mathcal{C}[se_2] \equiv \{f(se'_1, \ldots, s\mathcal{C}'[se_2], \ldots, se'_n)\} \twoheadrightarrow st} \text{ ROR}$$

Proof (For Theorem 1 (Compositionality)). For the part $\subseteq$, we must prove that for any context $s\mathcal{C}[se] \twoheadrightarrow st \Rightarrow \exists st' \in [\![se]\!]$ such that $s\mathcal{C}[st'] \twoheadrightarrow st$. First, we distinguish the possible contexts $s\mathcal{C}$:

a) if $s\mathcal{C} = [\,]$ we must prove: $se \twoheadrightarrow st \Rightarrow \exists st' \in [\![se]\!]$ such that $st' \twoheadrightarrow st$. This is trivial taking $st' = st$, as we have $st \twoheadrightarrow st$ for any $st \in SCTerm_\emptyset$ by Lemma 6.

b) if $s\mathcal{C} = \{\ldots, h(\ldots, s\mathcal{C}', \ldots), \ldots\}$, then we must prove: $\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st \Rightarrow \exists st' \in [\![se]\!]$ such that $\{\ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots\} \twoheadrightarrow st$. We proceed by induction on the length K of the derivation for $\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st$.

- $K = 1$: if the derivation is $\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow \emptyset$ by rule **E** we can take any value st' in $[\![se]\!]$ and trivially make $\{\ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots\} \twoheadrightarrow \emptyset$.
  The derivation can not be done by rule **RR** because of the syntactic forms. Neither it can be done by **DC** or some of the other rules in a single step.
- $K > 1$: the derivation can be done by the following rules:
  - by **DC**, with the form:

$$\frac{\ldots, s\mathcal{C}'[se] \twoheadrightarrow st'', \ldots}{\{c(\ldots, s\mathcal{C}'[se], \ldots)\} \twoheadrightarrow \{c(\ldots, st'', \ldots)\} \equiv st}$$

    Either by IH or by case *a*) if $s\mathcal{C}' = []$, there exists $st' \in [\![se]\!]$ such that $s\mathcal{C}'[st'] \twoheadrightarrow st''$ and then we can build the derivation for $\{c(\ldots, s\mathcal{C}'[st'], \ldots)\} \twoheadrightarrow \{c(\ldots, st'', \ldots)\} \equiv st$.

∗ by **More**, with the form:

$$\frac{\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_1 \ldots \{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_n}{\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st}$$

Either by IH or by case *a*) if $s\mathcal{C}' = []$, there exists $st'_1, \ldots, st'_n \in [\![se]\!]$ such that $\{\ldots, h(\ldots, s\mathcal{C}'[st'_i], \ldots), \ldots\} \twoheadrightarrow st_i$ for each i. Now we define $st' = st'_1 \cup \ldots \cup st'_n$ so that $st'_i \subseteq st'$, and then by Lemma 13 we have $\{\ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots\} \twoheadrightarrow st_i$ for each i. Then we can build the derivation

$$\frac{\{\ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots\} \twoheadrightarrow st_1 \ldots \{\ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots\} \twoheadrightarrow st_n}{\{\ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots\} \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st}$$

∗ by **Less**, with the form:

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \ldots \{esa_m\} \twoheadrightarrow st_n}{\{ese_1, \ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m \equiv st}$$

taking $\{esa_1, \ldots, esa_m\} \subseteq \{ese_1, \ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots, ese_n\}$. If $h(\ldots, s\mathcal{C}'[se], \ldots) \notin \{esa_1, \ldots, esa_m\}$ the proof is trivial. In the other case, if $h(\ldots, s\mathcal{C}'[se], \ldots) = esa_i$ for some i, then either by IH or by case *a*) if $s\mathcal{C}' = []$, there exists $st' \in [\![se]\!]$ such that $s\mathcal{C}'[st'] \twoheadrightarrow st_i$ and we can build the derivation:

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \ldots \{h(\ldots, s\mathcal{C}'[st'], \ldots)\} \twoheadrightarrow st_i \ldots \{esa_m\} \twoheadrightarrow st_n}{\{ese_1, \ldots, h(\ldots, s\mathcal{C}'[st'], \ldots), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_i \cup \ldots \cup st_m \equiv st}$$

∗ by **ROR** with the form:

$$\frac{\ldots s\mathcal{C}'[se] \twoheadrightarrow \tilde{p}\theta \ldots \tilde{r}\theta \twoheadrightarrow st}{\{f(\ldots, s\mathcal{C}'[se], \ldots)\} \twoheadrightarrow st}$$

having $(f(\ldots, p, \ldots) \to r) \in \mathcal{P}$ and $\theta \in SCSubst_{\emptyset}$.
Either by IH or by case *a*) if $s\mathcal{C}' = []$, there exists $st' \in [\![se]\!]$ such that $s\mathcal{C}'[st'] \twoheadrightarrow \tilde{p}\theta$ and then we can build the proof for $\{f(\ldots, s\mathcal{C}'[st'], \ldots)\} \twoheadrightarrow st$.

For the part $\supseteq$ we must prove that for any context $s\mathcal{C}$ if $se \twoheadrightarrow st$ and $s\mathcal{C}[st] \twoheadrightarrow st'$ then $s\mathcal{C}[se] \twoheadrightarrow st'$. As before, we distinguish the possible contexts $s\mathcal{C}$:

a) if $s\mathcal{C} = [\,]$ we must prove: if $\exists st$ such that $se \twoheadrightarrow st$ and $st \twoheadrightarrow st'$, then $se \twoheadrightarrow st'$. By Lemma 8 $st \twoheadrightarrow st'$ implies $st' \sqsubseteq st$; on the other hand $se \sqsubseteq se$. So we can apply Prop. 1 to obtain $se \twoheadrightarrow st'$.
b) if $s\mathcal{C} = \{\ldots, h(\ldots, s\mathcal{C}', \ldots), \ldots\}$, then we must prove: if $\exists st \in [\![se]\!]$ such that $\{\ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots\} \twoheadrightarrow st'$, then $\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st'$. We proceed by induction on the length K of the derivation for $\{\ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots\} \twoheadrightarrow st'$:
   - $K = 1$: if the derivation is $\{\ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots\} \twoheadrightarrow \emptyset$ by rule **E**, trivially we can derive $\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow \emptyset$ by the same rule **E**. The derivation can not be done by rule **RR** because of the syntactic forms. Neither it can be done by **DC** or some of the other rules in a single step.
   - $K > 1$: the derivation can be done by the following rules:
     ∗ by **DC**, with the form:

$$\frac{\ldots, s\mathcal{C}'[st] \twoheadrightarrow st'', \ldots}{\{c(\ldots, s\mathcal{C}'[st], \ldots)\} \twoheadrightarrow \{c(\ldots, st'', \ldots)\} \equiv st'}$$

     Either by IH or by case *a*) if $s\mathcal{C}' = []$, $s\mathcal{C}'[se] \twoheadrightarrow st'$ and then we can build the derivation for $\{c(\ldots, s\mathcal{C}'[se], \ldots)\} \twoheadrightarrow \{c(\ldots, st'', \ldots)\} \equiv st'$.
     ∗ by **More**, with the form:

$$\frac{\{\ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots\} \twoheadrightarrow st_1 \ldots \{\ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots\} \twoheadrightarrow st_n}{\{\ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots\} \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st'}$$

     Either by IH or by case *a*) if $s\mathcal{C}' = []$, $\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_i$ for each i and we can build the derivation

$$\frac{\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_1 \ldots \{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_n}{\{\ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots\} \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st'}$$

∗ by **Less**, with the form:

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \ldots \{esa_m\} \twoheadrightarrow st_n}{\{ese_1, \ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m \equiv st'}$$

taking $\{esa_1, \ldots, esa_m\} \subseteq \{ese_1, \ldots, h(\ldots, s\mathcal{C}'[st], \ldots), \ldots, ese_n\}$. If $h(\ldots, s\mathcal{C}'[st], \ldots) \notin \{esa_1, \ldots, esa_m\}$ the proof is trivial. In the other case, if $h(\ldots, s\mathcal{C}'[st], \ldots) = esa_i$ for some i, then either by IH or by case *a*) if $s\mathcal{C}' = []$, we have $\{h(\ldots, s\mathcal{C}'[se], \ldots) \twoheadrightarrow st_i$ and we can build the derivation:

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \ldots \{h(\ldots, s\mathcal{C}'[se], \ldots)\} \twoheadrightarrow st_i \ldots \{esa_m\} \twoheadrightarrow st_n}{\{ese_1, \ldots, h(\ldots, s\mathcal{C}'[se], \ldots), \ldots, ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_i \cup \ldots \cup st_m \equiv st'}$$

∗ by **ROR** with the form:

$$\frac{\ldots s\mathcal{C}'[st] \twoheadrightarrow \tilde{p}\theta \ldots \tilde{r}\theta \twoheadrightarrow st'}{\{f(\ldots, s\mathcal{C}'[st], \ldots)\} \twoheadrightarrow st'}$$

having $(f(\ldots, p, \ldots) \to r) \in \mathcal{P}$ and $\theta \in SCSubst_{\emptyset}$.
Either by IH or by case *a*) if $s\mathcal{C}' = []$, we have $s\mathcal{C}'[se] \twoheadrightarrow \tilde{p}\theta$ and then we can build the proof for $\{f(\ldots, s\mathcal{C}'[se], \ldots)\} \twoheadrightarrow st'$.

Proof (For Proposition 3 (Closedness under substitutions)). First we prove *a)* by induction on the structure of $se \twoheadrightarrow st$. Concerning the base cases:

E Then we have $se \twoheadrightarrow \emptyset \equiv st$, but then $se\theta \twoheadrightarrow \emptyset \equiv \emptyset\theta \equiv st\theta$ by E too.
RR Then $se \equiv \{X\} \twoheadrightarrow \{X\} \equiv st$, but then $se\theta \equiv \theta(X) \in SCTerm$, as $\theta \in SCSubst$, and then $se\theta \equiv \theta(X) \twoheadrightarrow \theta(X) \equiv st\theta$ by Lemma 6.
DC Then $se \equiv \{c\} \twoheadrightarrow \{c\} \equiv st$, but then $se\theta \equiv \{c\} \twoheadrightarrow \{c\} \equiv \{c\}\theta \equiv st\theta$, by DC.

Concerning the inductive steps:

DC Then we can apply the IH to each $se_i \twoheadrightarrow st_i$ to get $se_i\theta \twoheadrightarrow st_i\theta$, and build

$$\frac{se_1\theta \twoheadrightarrow st_1\theta \quad \ldots \quad se_n\theta \twoheadrightarrow st_n\theta}{se\theta \equiv \{c(se_1\theta, \ldots, se_n\theta)\} \twoheadrightarrow \{c(st_1\theta, \ldots, st_n\theta)\} \equiv st\theta}\ \text{DC}$$

More Then we can apply the IH to each $se \twoheadrightarrow st_i$ to get $se\theta \twoheadrightarrow st_i\theta$, and build

$$\frac{se\theta \twoheadrightarrow st_1\theta \quad \ldots \quad se\theta \twoheadrightarrow st_n\theta}{se\theta \twoheadrightarrow st_1\theta \cup \ldots \cup st_n\theta \equiv st\theta}\ \textsc{More}$$

Less Then we can apply the IH to each $\{esa_i\} \twoheadrightarrow st_i$ to get $\{esa_i\}\theta \twoheadrightarrow st_i\theta$. But then $[\![\{esa_i\}\theta]\!] = [\![esa_i\theta]\!] \subseteq [\![se\theta]\!]$, by Lemma 11. So $\forall i, se\theta \twoheadrightarrow st_i\theta$, and we can build the following proof

$$\frac{se\theta \twoheadrightarrow st_1\theta \quad \ldots \quad se\theta \twoheadrightarrow st_m\theta}{se\theta \twoheadrightarrow st_1\theta \cup \ldots \cup st_m\theta \equiv st\theta}\ \textsc{More}$$

ROR Then we have

$$\frac{se_1 \twoheadrightarrow \tilde{p_1}\mu \quad \ldots \quad se_n \twoheadrightarrow \tilde{p_n}\mu \quad \tilde{r}\mu \twoheadrightarrow st}{se \equiv \{f(se_1, \ldots, se_n)\} \twoheadrightarrow st}\ \text{ROR}$$

Then we can apply the IH to each $se_i \twoheadrightarrow \tilde{p_i}\mu$ to get that $se_i\theta \twoheadrightarrow \tilde{p_i}\mu\theta$, and to $\tilde{r}\mu \twoheadrightarrow st$ to get that $\tilde{r}\mu\theta \twoheadrightarrow st\theta$. Besides as $\mu, \theta \in SCSubst$ then $\mu\theta \in SCSubst$ too, and so we can do

$$\frac{se_1\theta \twoheadrightarrow \tilde{p_1}\mu\theta \quad \ldots \quad se_n\theta \twoheadrightarrow \tilde{p_n}\mu\theta \quad \tilde{r}\mu\theta \twoheadrightarrow st\theta}{se\theta \equiv \{f(se_1\theta, \ldots, se_n\theta)\} \twoheadrightarrow st\theta}\ \text{ROR}$$

Now we can prove *b)* also. Assume $st\sigma \twoheadrightarrow st'$ for some $st' \in SCTerm$, then by Lemma 1 $\exists\theta \in [\![\sigma]\!]$ such that $st\theta \twoheadrightarrow st'$. But $\theta \in [\![\sigma]\!]$ implies $\theta \in SCSubst$ by definition and $\theta \trianglelefteq \sigma$ by Lemma 12. Therefore $st\theta \in SCTerm$ and so $st\theta \twoheadrightarrow st'$ implies $st' \sqsubseteq st\theta$ by Lemma 8, and as $se \twoheadrightarrow st$ implies $se\theta \twoheadrightarrow st\theta$ by part *a)*, then $se\theta \twoheadrightarrow st'$ by Prop. 1. But then we can apply Prop. 2 over $\theta \trianglelefteq \sigma$ and $se\theta \twoheadrightarrow st'$ to get $se\sigma \twoheadrightarrow st'$.

A.3 For Subsection 3.3

Definition 4. *For any non empty and finite set* $\{\theta_1,\ldots,\theta_n\} \subseteq SCSubst$ *we define* $\bigcup\{\theta_1,\ldots,\theta_n\} \in SCSubst$ *as* $\bigcup\{\theta_1,\ldots,\theta_n\}(X) = \theta_1(X) \cup \ldots \cup \theta_n(X)$.

Lemma 14. *For any non empty and finite set* $\{\theta_1,\ldots,\theta_n\} \subseteq SCSubst$.

i) $dom(\bigcup\{\theta_1,\ldots,\theta_n\}) = \bigcup_i dom(\theta_i)$.
ii) $\forall\theta_i \in \{\theta_1,\ldots,\theta_n\}$ *we have* $\theta_i \trianglelefteq \bigcup\{\theta_1,\ldots,\theta_n\}$.
iii) If $\forall\theta_i \in \{\theta_1,\ldots,\theta_n\}, \theta_i \in [\![\sigma]\!]$ *for some* $\sigma \in SSubst$ *then* $\bigcup\{\theta_1,\ldots,\theta_n\} \in [\![\sigma]\!]$ *too.*

Proof.

i) Given some $X \in \mathcal{V}$, if $X \in \bigcup_i dom(\theta_i)$ then $\exists\theta_i \in \{\theta_1,\ldots,\theta_n\}$ such that $X \in dom(\theta_i)$, hence $\theta_i(X) \not\equiv \{X\}$. But then $\bigcup\{\theta_1,\ldots,\theta_n\}(X) \equiv \theta_1(X) \cup \ldots \cup \theta_n(X) \not\equiv \{X\}$, hence $X \in dom(\bigcup\{\theta_1,\ldots,\theta_n\})$.
On the other hand if $X \notin \bigcup_i dom(\theta_i)$ then $\forall\theta_i \in \{\theta_1,\ldots,\theta_n\}$ we have $\theta_i(X) \equiv \{X\}$ and so $\bigcup\{\theta_1,\ldots,\theta_n\}(X) \equiv \theta_1(X) \cup \ldots \cup \theta_n(X) \equiv \{X\} \cup \ldots \cup \{X\} = \{X\}$, hence $X \notin dom(\bigcup\{\theta_1,\ldots,\theta_n\})$.
ii) Given any $X \in \mathcal{V}$ we have $\theta_i(X) \subseteq \bigcup\{\theta_1,\ldots,\theta_n\}(X)$ by definition, hence $[\![\theta_i(X)]\!] \subseteq [\![\bigcup\{\theta_1,\ldots,\theta_n\}(X)]\!]$ by Lemma 7 and Prop. 1.
iii) Given any $X \in \mathcal{V}$ we can build

$$\frac{\dfrac{\theta_1 \in [\![\sigma]\!]}{\sigma(X) \twoheadrightarrow \theta_1(X)} \quad \ldots \quad \dfrac{\theta_n \in [\![\sigma]\!]}{\sigma(X) \twoheadrightarrow \theta_n(X)}}{\sigma(X) \twoheadrightarrow \theta_1(X) \cup \ldots \cup \theta_n(X) \equiv \bigcup\{\theta_1,\ldots,\theta_n\}(X)}\ \textsc{More}$$

Proof (For Lemma 1). We proceed by a case distinction over se. If $se \equiv \{X\}$ for some $X \in dom(\sigma)$: Then the hypothesis is $\sigma(X) \twoheadrightarrow st$, and we can define $\theta \in SCSubst$ as

$$\theta(Y) = \begin{cases} st & \text{if } Y \equiv X \\ \emptyset & \text{if } Y \in dom(\sigma) \setminus \{X\} \\ \{Y\} & \text{otherwise} \end{cases}$$

But then $\theta \in [\![\sigma]\!]$, because given any $Y \in \mathcal{V}$ we have the following possibilities:

a) $Y \equiv X$: Then $\sigma(Y) \twoheadrightarrow st \equiv \theta(Y)$ by hypothesis.
b) $Y \in dom(\sigma) \setminus \{X\}$: Then $\sigma(Y) \twoheadrightarrow \emptyset \equiv \theta(Y)$, using rule E.
b) $Y \notin dom(\sigma)$: Then $\sigma(Y) \equiv \{Y\} \twoheadrightarrow \{Y\} \equiv \theta(Y)$, using rule RR.

Besides, $\theta(X) \equiv st \twoheadrightarrow st$ by Lemma 6, so we are done.

On the other hand, if $se \equiv \{X\}$ for some $X \in \mathcal{V} \setminus dom(\sigma)$: Then the hypothesis is $\sigma(X) \equiv \{X\} \twoheadrightarrow st$, so if $\overline{Y} = dom(\sigma)$ then we can take $\theta = [\overline{Y/\emptyset}]$ for which is very easy to check $\theta \in [\![\sigma]\!]$, in a similar way to the previous case. But then $\theta(X) \equiv \{X\} \twoheadrightarrow st$ by hypothesis.

Otherwise we proceed by induction on the structure of $se\sigma \twoheadrightarrow st$. Concerning the base cases:

E Then $st \equiv \emptyset$ and so if $\overline{Y} = dom(\sigma)$ then we can take $\theta = [\overline{Y/\emptyset}]$ for which is very easy to check $\theta \in [\![\sigma]\!]$. Then $se\theta \twoheadrightarrow \emptyset \equiv st$ by E, so we are done.
RR Then $se\sigma \equiv \{Y\} \twoheadrightarrow \{Y\} \equiv st$, and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{Y\}$, hence $\forall ese \in se, ese \in \mathcal{V}$. Besides $se\sigma \equiv \{Y\}$ implies $se \not\equiv \emptyset$, therefore $\exists Z \in se \cap \mathcal{V}$ such that $\{Z\}\sigma \equiv Z\sigma \equiv \{Y\} \twoheadrightarrow st$ by hypothesis. But then by the proof of the cases when $se \equiv \{X\}$ we get some $\theta \in [\![\sigma]\!]$ such that $Z\theta \equiv \{Z\}\theta \twoheadrightarrow st$. But then $st \in [\![Z\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.
DC Then $se\sigma \equiv \{c\} \twoheadrightarrow \{c\} \equiv st$, and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{c\}$. We have two possibilities:
a) $se \cap \mathcal{V} \neq \emptyset$: Then given some $Y \in se \cap \mathcal{V}$ we have $\{Y\}\sigma \equiv Y\sigma \equiv \{c\} \twoheadrightarrow st$ by hypothesis. But then by the proof of the cases when $se \equiv \{X\}$ we get some $\theta \in [\![\sigma]\!]$ such that $Y\theta \equiv \{Y\}\theta \twoheadrightarrow st$. But then $st \in [\![Y\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.

b) $se \cap \mathcal{V} = \emptyset$: Then $\forall ese \in se, ese\sigma \equiv \{c\}$ implies $\forall ese \in se, ese \equiv \{c\}$. Now if $\overline{Y} = dom(\sigma)$ we can take $\theta = [\overline{Y/\emptyset}]$ for which is very easy to check $\theta \in [\![\sigma]\!]$. Besides $se\sigma \equiv \{c\}$ implies $se \not\equiv \emptyset$, therefore $\exists ese \in se, ese\theta \equiv \{c\}\theta \equiv \{c\} \twoheadrightarrow st$ by hypothesis. But then $st \in [\![ese\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.

Concerning the inductive steps:

DC Then we have

$$\frac{se_1 \twoheadrightarrow st_1 \quad \ldots \quad se_n \twoheadrightarrow st_n}{se\sigma \equiv \{c(se_1, \ldots, se_n)\} \twoheadrightarrow \{c(st_1, \ldots, st_n)\} \equiv st} \text{ DC}$$

and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{c(se_1, \ldots, se_n)\}$. We have two possibilities:

a) $se \cap \mathcal{V} \neq \emptyset$: Then given some $Y \in se \cap \mathcal{V}$ we have $\{Y\}\sigma \equiv Y\sigma \equiv \{c(se_1, \ldots, se_n)\} \twoheadrightarrow st$ by hypothesis. But then by the proof of the cases when $se \equiv \{X\}$ we get some $\theta \in [\![\sigma]\!]$ such that $Y\theta \equiv \{Y\}\theta \twoheadrightarrow st$. But then $st \in [\![Y\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.

b) $se \cap \mathcal{V} = \emptyset$: Then $se\sigma \equiv \{c(se_1, \ldots, se_n)\}$ implies $se \not\equiv \emptyset$. Therefore $\exists ese \in se = (se \setminus \mathcal{V})$ such that $ese\sigma \equiv \{c(se_1, \ldots, se_n)\}$, which implies $ese \equiv c(se'_1, \ldots, se'_n)$ such that $\forall i, se'_i\sigma \equiv se_i \twoheadrightarrow st_i$, to which we can apply the IH or the proof for the cases when $se \equiv \{X\}$ to get some $\theta_i \in [\![\sigma]\!]$ such that $se'_i\theta_i \twoheadrightarrow st_i$. But then we can take $\theta = \bigcup\{\theta_1, \ldots, \theta_n\} \in [\![\sigma]\!]$ by Lemma 14, to get $\forall i, \theta_i \trianglelefteq \theta$ by Lemma 14, hence $\forall i, se'_i\theta \twoheadrightarrow st_i$ by Prop. 2 and we can build the following proof:

$$\frac{se'_1\theta \twoheadrightarrow st_1 \quad \ldots \quad se'_n\theta \twoheadrightarrow st_n}{ese\theta \equiv \{c(se'_1\theta, \ldots, se'_n\theta)\} \twoheadrightarrow \{c(st_1, \ldots, st_n)\} \equiv st} \text{ DC}$$

But then $st \in [\![ese\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.

More Then we have

$$\frac{se\sigma \twoheadrightarrow st_1 \quad \ldots \quad se\sigma \twoheadrightarrow st_n}{se\sigma \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st} \textsc{More}$$

But then we can apply the IH or the proof for the cases when $se \equiv \{X\}$ to each $se\sigma \twoheadrightarrow st_i$ to get some $\theta_i \in [\![\sigma]\!]$ such that $se\theta_i \twoheadrightarrow st_i$. But then we can take $\theta = \bigcup\{\theta_1, \ldots, \theta_n\} \in [\![\sigma]\!]$ by Lemma 14, to get $\forall i, \theta_i \trianglelefteq \theta$ by Lemma 14, hence $\forall i, se\theta \twoheadrightarrow st_i$ by Lemma 2 and we can build the following proof:

$$\frac{se\theta \twoheadrightarrow st_1 \quad \ldots \quad se\theta \twoheadrightarrow st_n}{se\theta \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st} \textsc{More}$$

Less Then we have

$$\frac{\{esa_1\} \twoheadrightarrow st_1 \quad \ldots \quad \{esa_n\} \twoheadrightarrow st_m}{se\sigma \twoheadrightarrow st_1 \cup \ldots \cup st_m \equiv st} \textsc{Less}$$

for some $\{esa_1, \ldots, esa_m\} \subseteq se\sigma$ and $\sharp se\sigma \geq 2$. Then for any $esa_i \in \{esa_1, \ldots, esa_m\} \subseteq se\sigma$ we have two possibilities:

a) $esa_i \in \sigma(X)$ for some $X \in se$: Then $st_i \in [\![\{esa_i\}]\!] \subseteq [\![\sigma(X)]\!] = [\![\{X\}\sigma]\!]$ by Lemma 7 and Prop. 1, as $\{esa_i\} \subseteq \sigma(X)$. Therefore by the proof of the case when $se \equiv \{X\}$ we get some $\theta_i \in [\![\sigma]\!]$ such that $\{X\}\theta_i \twoheadrightarrow st_i$. But then $X \in se$ implies $st_i \in [\![\{X\}\theta_i]\!] = [\![X\theta_i]\!] \subseteq [\![se\theta_i]\!]$ by Lemma 11.

b) $esa_i \equiv h(se_1\sigma, \ldots, se_n\sigma)$ for some $h(se_1, \ldots, se_n) \in se$. But then $\{h(se_1, \ldots, se_n)\}\sigma \equiv \{h(se_1\sigma, \ldots, se_n\sigma)\} \equiv \{esa_i\} \twoheadrightarrow st_i$ by hypothesis, and we can apply the IH to get some $\theta_i \in [\![\sigma]\!]$ such that $\{h(se_1, \ldots, se_n)\}\theta_i \twoheadrightarrow st_i$. But then $h(se_1, \ldots, se_n) \in se$ implies $st_i \in [\![\{h(se_1, \ldots, se_n)\}\theta_i]\!] = [\![h(se_1, \ldots, se_n)\theta_i]\!] \subseteq [\![se\theta_i]\!]$ by Lemma 11.

Therefore $\forall i \in \{1, \ldots, m\}, \exists \theta_i \in [\![\sigma]\!], se\theta_i \twoheadrightarrow st_i$. But then we can take $\theta = \bigcup\{\theta_1, \ldots, \theta_m\} \in [\![\sigma]\!]$ by Lemma 14, to get $\forall i, \theta_i \trianglelefteq \theta$ by Lemma 14, hence $\forall i, se\theta \twoheadrightarrow st_i$ by Prop. 2 and we can build the following proof:

$$\frac{se\theta \twoheadrightarrow st_1 \quad \ldots \quad se\theta \twoheadrightarrow st_m}{se\theta \twoheadrightarrow st_1 \cup \ldots \cup st_m \equiv st} \textsc{More}$$

ROR Then we have

$$\frac{se_1 \twoheadrightarrow \tilde{p_1}\mu \quad \ldots \quad se_n \twoheadrightarrow \tilde{p_n}\mu \quad \tilde{r}\mu \twoheadrightarrow st}{se\sigma \equiv \{f(se_1, \ldots, se_n)\} \twoheadrightarrow st} \text{ ROR}$$

and so by Lemma 10 $\forall ese \in se, ese\sigma \equiv \{f(se_1, \ldots, se_n)\}$. We have two possibilities:

a) $se \cap \mathcal{V} \neq \emptyset$: Then given some $Y \in se \cap \mathcal{V}$ we have $\{Y\}\sigma \equiv Y\sigma \equiv \{f(se_1, \ldots, se_n)\} \twoheadrightarrow st$ by hypothesis. But then by the proof of the cases when $se \equiv \{X\}$ we get some $\theta \in [\![\sigma]\!]$ such that $Y\theta \equiv \{Y\}\theta \twoheadrightarrow st$. But then $st \in [\![Y\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.

b) $se \cap \mathcal{V} = \emptyset$: Then $se\sigma \equiv \{f(se_1, \ldots, se_n)\}$ implies $se \not\equiv \emptyset$. Therefore $\exists ese \in se = (se \setminus \mathcal{V})$ such that $ese\sigma \equiv \{f(se_1, \ldots, se_n)\}$, which implies $ese \equiv f(se'_1, \ldots, se'_n)$ such that $\forall i, se'_i\sigma \equiv se_i \twoheadrightarrow \tilde{p_i}\mu$, to which we can apply the IH or the proof for the cases when $se \equiv \{X\}$ to get some $\theta_i \in [\![\sigma]\!]$ such that $se'_i\theta_i \twoheadrightarrow \tilde{p_i}\mu$. But then we can take $\theta = \bigcup\{\theta_1, \ldots, \theta_n\} \in [\![\sigma]\!]$ by Lemma 14, to get $\forall i, \theta_i \trianglelefteq \theta$ by Lemma 14, hence $\forall i, se'_i\theta \twoheadrightarrow \tilde{p_i}\mu$ by Prop. 2 and we can build the following proof:

$$\frac{se'_1\theta \twoheadrightarrow \tilde{p_1}\mu \quad \ldots \quad se'_n\theta \twoheadrightarrow \tilde{p_n}\mu \quad \dfrac{}{\tilde{r}\mu \twoheadrightarrow st}\,hypothesis}{ese\theta \equiv \{f(se'_1\theta, \ldots, se'_n\theta)\} \twoheadrightarrow st}\ \text{ROR}$$

But then $st \in [\![ese\theta]\!] \subseteq [\![se\theta]\!]$ by Lemma 11, and so $se\theta \twoheadrightarrow st$.

Lemma 15. *Under any program, $\forall e, e' \in Exp$ if $e \to e'$ then $[\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$.*

Proof. Assume $e \to e'$. If the step was performed at the root then we have $e \equiv f(p_1, \ldots, p_n)\sigma \to r\sigma \equiv e'$ for some rule $(f(p_1, \ldots, p_n) \to r) \in \mathcal{P}$. Assume $\tilde{e'} \equiv \widetilde{r\sigma} \twoheadrightarrow st$ for some $st \in SCTerm$, then we have $\tilde{r}\tilde{\sigma} \equiv \widetilde{r\sigma} \twoheadrightarrow st$ by Lemma 9, but then $\exists \theta \in [\![\tilde{\sigma}]\!]$ such that $\tilde{r}\theta \twoheadrightarrow st$, by Lemma 1. Besides for each p_i we have $p_i \in CTerm$ but then is easy to prove that $\tilde{p_i} \in SCTerm$ and so $\tilde{p_i}\theta \in SCTerm$, because $\theta \in [\![\tilde{\sigma}]\!]$ implies $\theta \in SCSubst$. But then by Lemma 6 we have $\tilde{p_i}\theta \twoheadrightarrow \tilde{p_i}\theta$, and then we can build

$$\frac{\tilde{p_1}\theta \twoheadrightarrow \tilde{p_1}\theta \quad \ldots \quad \tilde{p_n}\theta \twoheadrightarrow \tilde{p_n}\theta \quad \tilde{r}\theta \twoheadrightarrow st}{\widetilde{f(p_1, \ldots, p_n)}\theta \equiv \{f(\tilde{p_1}\theta, \ldots, \tilde{p_n}\theta)\} \twoheadrightarrow st}\ \text{ROR}$$

But $\theta \in [\![\tilde{\sigma}]\!]$ implies $\theta \trianglelefteq \tilde{\sigma}$ by Lemma 12, and so $\widetilde{f(p_1, \ldots, p_n)}\theta \twoheadrightarrow st$ implies $\widetilde{f(p_1, \ldots, p_n)}\tilde{\sigma} \twoheadrightarrow st$ by Prop. 2. But $\widetilde{f(p_1, \ldots, p_n)}\tilde{\sigma} \equiv \widetilde{f(p_1, \ldots, p_n)}\sigma$ by Lemma 9, therefore $\tilde{e} \equiv \widetilde{f(p_1, \ldots, p_n)}\sigma \equiv \widetilde{f(p_1, \ldots, p_n)}\tilde{\sigma} \twoheadrightarrow st$, and so we have proved that $[\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$.

Otherwise if the step was not performed at the root we have $e \equiv \mathcal{C}[f(\overline{p})\sigma] \to \mathcal{C}[r\sigma] \equiv e'$, where $f(\overline{p})\sigma \to r\sigma$ has been performed at the root and so by the proof of the other case $[\![r\sigma]\!] \subseteq [\![f(\overline{p})\sigma]\!]$. Assume $\tilde{e'} \equiv \widetilde{\mathcal{C}[r\sigma]} \twoheadrightarrow st$ for some $st \in SCTerm$, then we can chain:

$$\begin{array}{ll}
[\![\tilde{e'}]\!] = [\![\widetilde{\mathcal{C}[r\sigma]}]\!] & \\
= [\![\tilde{\mathcal{C}}[\widetilde{r\sigma}]]\!] & \text{by Lemma 9} \\
= \bigcup_{st \in [\![\widetilde{r\sigma}]\!]} [\![\tilde{\mathcal{C}}[st]]\!] & \text{by Theorem 1} \\
\subseteq \bigcup_{st \in [\![\widetilde{f(\overline{p})\sigma}]\!]} [\![\tilde{\mathcal{C}}[st]]\!] & \text{as } [\![r\sigma]\!] \subseteq [\![f(\overline{p})\sigma]\!] \\
= [\![\tilde{\mathcal{C}}[\widetilde{f(\overline{p})\sigma}]]\!] & \text{by Theorem 1} \\
= [\![\widetilde{\mathcal{C}[f(\overline{p})\sigma]}]\!] = [\![\tilde{e}]\!] & \text{by Lemma 9}
\end{array}$$

Proof (For Proposition 4). A simple induction on the length of the derivation $e \to^* e'$. The base case is trivial and the inductive step is straightforward too, using Lemma 15 for the first step, applying the IH to the others and using the transitivity of set inclusions to chain those results.

Lemma 16. *Under any program and $\forall st, st' \in SCTerm, est, est' \in ESCTerm, e \in Exp$:*

- *If $st \sqsubseteq st'$ and $st' \lessdot e$ then $st \lessdot e$.*
- *If $est \sqsubseteq est'$ and $est' \lessdot e$ then $est \lessdot e$.*

Proof. We proceed by induction on the structure of st' and est'. The base cases are the following:

- $st' \equiv \emptyset$: Then $st \sqsubseteq st'$ implies $st \equiv \emptyset$, but then $st \equiv \emptyset \lessdot e$.
- $est' \equiv X \in \mathcal{V}$: Then $est \sqsubseteq est'$ implies $est \equiv X \equiv est'$, but then $est \equiv est' \lessdot e$ by hypothesis.
- $est' \equiv c \in CS^0$: Then $est \sqsubseteq est'$ implies $est \equiv c \equiv est'$, but then $est \equiv est' \lessdot e$

Concerning the inductive step:

- $st' \not\equiv \emptyset$: Then $st \sqsubseteq st'$ implies that $\forall est \in st, \exists est' \in st'$ such that $est \sqsubseteq est'$. But as $st' \lessdot e$, then for that est' we have $est' \lessdot e$ too. So we can apply the IH with $est \sqsubseteq est'$ and $est' \lessdot e$ to get that $est \lessdot e$, hence $\forall est \in st, est \lessdot e$, that is, $st \lessdot e$.
- $est' \equiv c(st'_1, \ldots, st'_n)$: Then $est \sqsubseteq est'$ implies $est \equiv c(st_1, \ldots, st_n)$ such that $\forall i, st_i \sqsubseteq st'_i$, and $est' \lessdot e$ implies $e \rightarrow^* c(e_1, \ldots, e_n)$ such that $\forall i, st'_i \lessdot e_i$. But then we can apply the IH to each $st_i \sqsubseteq st'_i$ and $st'_i \lessdot e_i$ to get $st_i \lessdot e_i$, hence $est \lessdot e$.

Lemma 17. *Under any program and $\forall \theta \in SCSubst, \sigma \in Subst$ if $dom(\theta) = dom(\sigma)$ and $\forall X \in dom(\theta), \theta(X) \lessdot \sigma(X)$, then $\theta \lessdot \sigma$.*

Proof. Given $X \in \mathcal{V}$, if $X \in dom(\theta)$ then $\theta(X) \lessdot \sigma(X)$ by hypothesis. Otherwise $X \notin dom(\theta) = dom(\sigma)$, therefore $\theta(X) \equiv X \lessdot X \equiv \sigma(X)$, as $X \rightarrow^0 X$.

Lemma 18. *Under any program and $\forall e \in Exp, p \in CTerm$ linear and $\theta \in SCSubst$ such that $dom(\theta) \subseteq var(p)$ we have that $\tilde{p}\theta \lessdot e$ implies that $\exists \sigma \in Subst$ such that $dom(\sigma) = dom(\theta)$, $\theta \lessdot \sigma$ and $e \rightarrow^* p\sigma$.*

Proof. We proceed by induction on the structure of p, the base cases are the following:

- $p \equiv X \in \mathcal{V}$: If $X \notin dom(\theta) \subseteq var(p) = \{X\}$ then $\theta = \epsilon$, hence $\tilde{p}\theta \lessdot e$ is equivalent to $\{X\} \lessdot e$, which implies $e \rightarrow^* X$. But then we can take $\sigma = \epsilon$ for which $dom(\sigma) = \emptyset = dom(\theta)$, $\theta = \epsilon \lessdot \epsilon = \sigma$ (because $\epsilon \lessdot \epsilon$ by Lemma 17) and $e \rightarrow^* X \equiv X\epsilon \equiv p\sigma$.
  On the other hand if $X \in dom(\theta) \subseteq var(p) = \{X\}$ then $dom(\theta) = \{X\}$ and $\tilde{p}\theta \lessdot e$ is equivalent to $\theta(X) \lessdot e$. But then we can take $\sigma = [X/e]$ for which $dom(\theta) = \{X\} = dom(\sigma)$ and $\theta(X) \lessdot e \equiv \sigma(X)$, therefore $\theta \lessdot \sigma$ by Lemma 17. Besides $e \rightarrow^0 e \equiv X[X/e] \equiv p\sigma$, so we are done.
- $p \equiv c \in CS^0$: Then as $dom(\theta) \subseteq var(p) = \emptyset$ we have $\theta = \epsilon$, hence $\tilde{p}\theta \lessdot e$ is equivalent to $\{c\} \lessdot e$, which implies $e \rightarrow^* c$. But then we can take $\sigma = \epsilon$ for which $dom(\sigma) = \emptyset = dom(\theta)$, $\theta = \epsilon \lessdot \epsilon = \sigma$ (because $\epsilon \lessdot \epsilon$ by Lemma 17) and $e \rightarrow^* c \equiv c\epsilon \equiv p\sigma$.

Concerning the inductive step, this happens when $p \equiv c(p_1, \ldots, p_n)$. Then as p is linear and $dom(\theta) \subseteq var(p)$, if $\forall i \in \{1, \ldots, n\}$ we define $\theta_i = \theta|_{var(p_i)}$, then $\theta = \theta_1 \uplus \ldots \uplus \theta_n$. Besides the hypothesis $\tilde{p}\theta \lessdot e$ is equivalent to $\{c(\tilde{p_1}, \ldots, \tilde{p_n})\}\theta \equiv \{c(\tilde{p_1}\theta, \ldots, \tilde{p_n}\theta)\} \equiv \{c(\tilde{p_1}\theta_1, \ldots, \tilde{p_n}\theta_n)\} \lessdot e$ (as $var(e_i) = var(\tilde{e_i})$, by Prop. 8), hence $e \rightarrow^* c(e_1, \ldots, e_n)$ such that $\forall i, \tilde{p_i}\theta_i \lessdot e_i$. But then by IH $\forall i, \exists \sigma_i \in Subst$ such that $dom(\sigma_i) = dom(\theta_i)$, $\theta_i \lessdot \sigma_i$ and $e_i \rightarrow^* p_i\sigma_i$. But $\forall i, dom(\sigma_i) = dom(\theta_i) \subseteq var(p_i)$ hence as p is linear then $\sigma = \sigma_1 \uplus \ldots \uplus \sigma_n$ is correctly defined and $\theta \lessdot \sigma$ holds, because $dom(\sigma) = \bigcup_i dom(\sigma_i) = \bigcup_i dom(\theta_i) = dom(\theta)$ and given some $X \in dom(\theta)$ then $X \in dom(\theta_i)$ for some θ_i and then $\theta(X) \equiv \theta_i(X) \lessdot \sigma_i(X) \equiv \sigma(X)$, therefore we can apply Lemma 17. Finally we can chain $e \rightarrow^* c(e_1, \ldots, e_n) \rightarrow^* c(p_1\sigma_1, \ldots, p_n\sigma_n) \equiv c(p_1, \ldots, p_n)\sigma \equiv p\sigma$, so we are done.

Lemma 19. *Under any program and $\forall e, e' \in Exp, st \in SCTerm$ if $st \lessdot e'$ and $e \rightarrow^* e'$ then $st \lessdot e$.*

Proof. It is enough to check that $\forall est \in st, est \lessdot e$, and that happens because:

- If $est \equiv X \in \mathcal{V}$ then $est \in st$ and $st \lessdot e'$ implies $X \equiv est \lessdot e'$, and so $e' \rightarrow^* X$. But then $e \rightarrow^* e' \rightarrow^* X$, hence $X \equiv est \lessdot e$ too.
- If $est \equiv c(st_1, \ldots, st_n)$ then $est \in st$ and $st \lessdot e'$ implies $c(st_1, \ldots, st_n) \equiv est \lessdot e'$, and so $e' \rightarrow^* c(e_1, \ldots, e_n)$ such that $\forall i, st_i \lessdot e_i$. But then $e \rightarrow^* e' \rightarrow^* c(e_1, \ldots, e_n)$, hence $c(st_1, \ldots, st_n) \equiv est \lessdot e$ too.

Proof (For Lemma 2). We proceed by a case distinction over e.
If $e \equiv X \in \mathcal{V}$ then $\tilde{e} \equiv \{X\} \in SCTerm$, but as $\theta \in SCSubst$ then $\tilde{e}\theta \equiv \theta(X) \in SCTerm$, and so $\theta(X) \equiv \tilde{e}\theta \twoheadrightarrow st$ implies $st \sqsubseteq \theta(X)$, by Lemma 8. Besides $\theta \lessdot \sigma$ implies $\theta(X) \lessdot \sigma(X)$ by definition, therefore we can combine it with $st \sqsubseteq \theta(X)$ to get $st \lessdot \sigma(X) \equiv e\sigma$ by Lemma 16.
Otherwise we will prove the case when e is not restricted to be a variable by induction on the structure of $\tilde{e}\theta \twoheadrightarrow st$. The base cases are the following:

E Then $st \equiv \emptyset$, therefore for any $\sigma \in Subst$ we have $st \equiv \emptyset \lessdot e\sigma$.
RR Then $\tilde{e}\theta \equiv \{X\}$, but that implies $e \in \mathcal{V}$, so we can proceed like in the previous case, at the beginning of the proof.

DC If $e \in \mathcal{V}$ then we proceed like in the previous case. Otherwise $e \equiv c \in CS^0$ and $st \equiv \{c\}$. But then for any $\sigma \in Subst$ $e\sigma \equiv c \to^0 c$, therefore $st \equiv \{c\} \lessdot e\sigma$.

Regarding the inductive steps:

DC If $e \in \mathcal{V}$ then we proceed like in the previous case. Otherwise $e \equiv c(e_1, \ldots, e_n)$ and the proof has the following case:

$$\frac{\tilde{e_1}\theta \twoheadrightarrow st_1 \quad \ldots \quad \tilde{e_n}\theta \twoheadrightarrow st_n}{\tilde{e}\theta \equiv \{c(\tilde{e_1}\theta, \ldots, \tilde{e_n}\theta)\} \twoheadrightarrow \{c(st_1, \ldots, st_n)\} \equiv st} \text{ DC}$$

Now given some $\sigma \in Subst$ such that $\theta \lessdot \sigma$ we can apply the IH to each $\tilde{e_i}\theta \twoheadrightarrow st_i$ to get that $st_i \lessdot e_i\sigma$. But then $e\sigma \equiv c(e_1\sigma, \ldots, e_n\sigma) \to^0 c(e_1\sigma, \ldots, e_n\sigma)$, therefore $st \equiv \{c(st_1, \ldots, st_n)\} \lessdot e\sigma$.

More Then the proof has the following shape:

$$\frac{\tilde{e}\theta \twoheadrightarrow st_1 \quad \ldots \quad \tilde{e}\theta \twoheadrightarrow st_n}{\tilde{e}\theta \twoheadrightarrow st_1 \cup \ldots \cup st_n \equiv st} \text{ More}$$

Now given some $\sigma \in Subst$ such that $\theta \lessdot \sigma$ we can apply the IH to each $\tilde{e}\theta \twoheadrightarrow st_i$ to get $st_i \lessdot e\sigma$, which implies that $\forall i, \forall est \in st_i, est \lessdot e\sigma$, therefore $st \equiv st_1 \cup \ldots \cup st_n \lessdot e\sigma$.

Less If $e \in \mathcal{V}$ then we proceed like in the previous case. Otherwise $e \equiv h(e_1, \ldots, e_n)$, hence $\tilde{e}\theta \equiv \{h(\tilde{e_1}\theta, \ldots, \tilde{e_n}\theta)\}$ and therefore Less cannot have been applied.

ROR If $e \in \mathcal{V}$ then we proceed like in the previous case. Otherwise $e \equiv f(e_1, \ldots, e_n)$ and the proof has the following shape:

$$\frac{\tilde{e_1}\theta \twoheadrightarrow \tilde{p_1}\mu \quad \ldots \quad \tilde{e_n}\theta \twoheadrightarrow \tilde{p_n}\mu \quad \tilde{r}\mu \twoheadrightarrow st}{\tilde{e}\theta \equiv \{f(\tilde{e_1}\theta, \ldots, \tilde{e_n}\theta)\} \twoheadrightarrow st} \text{ ROR}$$

for some $(f(p_1, \ldots, p_n) \to r) \in \mathcal{P}$. But then $var(r) \subseteq var(f(p_1, \ldots, p_n))$ and so we may assume $dom(\mu) \subseteq var(f(p_1, \ldots, p_n))$ without loss of generality. Besides, as $f(p_1, \ldots, p_n)$ is linear, if for each $i \in \{1, \ldots, n\}$ we define $\mu_i = \mu|_{var(\tilde{p_i})} = \mu_{var(p_i)}$ (as $var(\tilde{p_i}) = var(p_i)$ by Prop. 8), then $\mu = \mu_1 \uplus \ldots \uplus \mu_n$ and $\forall i, \tilde{p_i}\mu \equiv \tilde{p_i}\mu_i$.
Now given some $\sigma \in Subst$ such that $\theta \lessdot \sigma$ we can apply the IH to each $\tilde{e_i}\theta \twoheadrightarrow \tilde{p_i}\mu$ to get $\forall i, \tilde{p_i}\mu_i \equiv \tilde{p_i}\mu \lessdot e_i\sigma$. But then by Lemma 18 we have that $\forall i, \exists \sigma'_i \in Subst$ such that $dom(\sigma'_i) = dom(\mu_i)$, $\mu_i \lessdot \sigma'_i$ and $e_i\sigma \to^* p_i\sigma'_i$. As $\forall i, dom(\sigma'_i) = dom(\mu_i) \subseteq var(p_i)$, this combined with the linearity of $f(p_1, \ldots, p_n)$ implies that $\sigma' = \sigma'_1 \uplus \ldots \uplus \sigma'_n$ is correctly defined. Furthermore, $\mu \lessdot \sigma'$ because $dom(\sigma') = \bigcup_i dom(\sigma'_i) = \bigcup_i dom(\mu_i) = dom(\mu)$ and given some $X \in dom(\mu)$ then $X \in dom(\mu_i)$ for some i and then $\mu(X) \equiv \mu_i(X) \lessdot \sigma'_i(X) \equiv \sigma'(X)$, as $\mu_i \lessdot \sigma'_i$, therefore we can apply Lemma 17.
Finally, we can use σ' to apply the IH to $\tilde{r}\mu \twoheadrightarrow st$, getting that $st \lessdot r\sigma'$. Besides $e\sigma \equiv f(e_1\sigma, \ldots, e_n\sigma) \to^* f(p_1\sigma'_1, \ldots, p_n\sigma'_n) \equiv f(p_1, \ldots, p_n)\sigma' \to r\sigma'$, so we can combine $st \lessdot r\sigma'$ with $e\sigma \to^* r\sigma'$ using Lemma 19 to get that $st \lessdot e\sigma$.

Proof (For Lemma 3 (Domination)). Assume $\tilde{e} \twoheadrightarrow st$, then as $\epsilon \lessdot \epsilon$ we can apply Lemma 2 using $\theta = \epsilon = \sigma$ to get $st \lessdot e\sigma \equiv e$. To prove the converse implication we will prove that $\forall e \in Exp, st \in SCTerm, est \in ESCTerm$ $st \lessdot e$, $st \lessdot e$ implies $\tilde{e} \twoheadrightarrow st$ and $est \lessdot e$ implies $\tilde{e} \twoheadrightarrow \{est\}$, by induction on the proof for $st \lessdot e$ and $est \lessdot e$. The base cases are the following:

- $st \equiv \emptyset$: Then $\tilde{e} \twoheadrightarrow \emptyset \equiv st$ by E.
- $est \equiv X$: Then $X \lessdot e$ implies $e \to^* X$, which implies $[\![\{X\}]\!] \subseteq [\![\tilde{e}]\!]$, by Prop. 4. But $\{X\} \in [\![\{X\}]\!] \subseteq [\![\tilde{e}]\!]$, that is, $\tilde{e} \twoheadrightarrow \{X\} \equiv \{est\}$.
- $est \equiv c$: Then $c \lessdot e$ implies $e \to^* c$, which implies $[\![\{c\}]\!] \subseteq [\![\tilde{e}]\!]$, by Prop. 4. But $\{c\} \in [\![\{c\}]\!] \subseteq [\![\tilde{e}]\!]$, that is, $\tilde{e} \twoheadrightarrow \{c\} \equiv \{est\}$.

Concerning the inductive steps:

- $st \not\equiv \emptyset$: Then $\sharp st > 0$ and we have the following possibilities:
  i) $\sharp st = 1$: Then $st \equiv \{est\} \lessdot e$ implies $est \lessdot e$ by definition, but then $\tilde{e} \twoheadrightarrow \{est\} \equiv st$ by IH.
  ii) $\sharp st > 1$: Then $st \equiv \{est_1, \ldots, est_n\}$ and $st \lessdot e$ implies $\forall st_i \in st, est_i \lessdot e$ by definition. But then we can apply the IH to each $est_i \lessdot e$ to get $\tilde{e} \twoheadrightarrow \{est_i\}$ by IH, and we can build the following proof:

$$\frac{\tilde{e} \twoheadrightarrow \{est_1\} \quad \ldots \quad \tilde{e} \twoheadrightarrow \{est_n\}}{\tilde{e} \twoheadrightarrow \{est_1\} \cup \ldots \cup \{est_n\} \equiv st} \text{ More}$$

– $est \equiv c(st_1, \ldots, st_n)$: Then $c(st_1, \ldots, st_n) \lessdot e$ implies $e \to^* c(e_1, \ldots, e_n)$ such that $\forall i \in \{1, \ldots, n\}, st_i \lessdot e_i$. Then, by Prop. 4, $e \to^* c(e_1, \ldots, e_n)$ implies $[\![\widetilde{c(e_1, \ldots, e_n)}]\!] \subseteq [\![\tilde{e}]\!]$. Besides we can apply the IH to each $st_i \lessdot e_i$ to get $\tilde{e_i} \twoheadrightarrow st_i$, and build the following proof:

$$\frac{\tilde{e_1} \twoheadrightarrow st_1 \quad \ldots \quad \tilde{e_n} \twoheadrightarrow st_n}{\widetilde{c(e_1, \ldots, e_n)} \equiv \{\widetilde{c(e_1, \ldots, e_n)}\} \twoheadrightarrow \{c(st_1, \ldots, st_n)\} \equiv \{est\}}\ \text{DC}$$

But then $\{est\} \in [\![\widetilde{c(e_1, \ldots, e_n)}]\!] \subseteq [\![\tilde{e}]\!]$ implies $\tilde{e} \twoheadrightarrow \{est\}$.

Proof (For Lemma 4). To prove this lemma we prove a slight generalization of it:

$\forall st \in SCTerm, est \in ESCTerm, e \in Exp$ if $st \lessdot e$ ($est \lessdot e$ resp.) then $\forall t \in flat(st)$ ($\forall t \in flat(est)$ resp.) we have $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.

We proceed by induction on the structure of $SExp$ and $ESExp$. Concerning the base cases:

– $st \equiv \emptyset$: Then $flat(st) = \{\bot\}$ and $e \to^0 e$, with $\bot \sqsubseteq |e|$, so we are done.
– $est \equiv X$: Then $flat(est) = \{X\}$ and $X \equiv est \lessdot e$ implies $e \to^* X$, with $X \sqsubseteq X \equiv |X|$, so we are done.
– $est \equiv c$: Then $flat(est) = \{c\}$ and $c \equiv est \lessdot e$ implies $e \to^* c$, with $c \sqsubseteq c \equiv |c|$, so we are done.

Concerning the inductive steps:

– $st \not\equiv \emptyset$: Then given $t \in flat(st) = \bigcup_{est \in st} flat(est)$ then $\exists est_i \in st$ such that $t \in flat(est_i)$. Besides $st \lessdot e$ implies $est_i \lessdot e$ by definition, and so we can apply the HI over $est_i \lessdot e$ and $t \in flat(est_i)$ to get that $e \to^* e'$ such that $t \sqsubseteq |e'|$.
– $est \equiv c(st_1, \ldots, st_n)$: Then given $t \in flat(est)$ it must be $t \equiv c(t_1, \ldots, t_n)$ such that $\forall i \in \{1, \ldots, n\}, t_i \in flat(st_i)$. Besides $est \lessdot e$ implies $e \to^* c(e_1, \ldots, e_n)$ such that $\forall i \in \{1, \ldots, n\}, st_i \lessdot e_i$, hence we can apply the IH to each $st_i \lessdot e_i$ and $t_i \in flat(st_i)$ to get $e_i \to^* e'_i$ such that $t_i \sqsubseteq |e'_i|$. But then $e \to^* c(e_1, \ldots, e_n) \to^* c(e'_1, \ldots, e'_n)$ with $t \equiv c(t_1, \ldots, t_n) \sqsubseteq c(|e'_1|, \ldots, |e'_n|) \equiv |c(e'_1, \ldots, e'_n)|$.

A.4 For Section 4

Proof (Prop. 5). The $\Rightarrow$-implication is trivial. For the $\Leftarrow$-implication, assume $[\![e]\!]_{S'} = [\![e']\!]_{S'}$ and let $st \equiv \{est_1, \ldots, est_n\} \in [\![e]\!]_S$. By definition of $[\![e]\!]_{S'}$, each $est_i \in [\![e]\!]_{S'}$, and therefore each $est_i \in [\![e']\!]_{S'}$, which means that there exist $st_i \in [\![e']\!]_S$ such that $est_i \in st_i$. But now, since $\{est_i\} \subseteq st_i$, we know from polarity (Prop. 1 that $\{est_i\} \in [\![e']\!]_S$, for each i, and polarity implies also $st \equiv \{est_1, \ldots, est_n\} \in [\![e']\!]_S$. We have then proved that $[\![e]\!]_S \subseteq [\![e']\!]_S$. The other inclusion holds similarly, arriving to the desired $[\![e]\!]_S = [\![e']\!]_S$.

Proof (Prop. 6).

(a) Full abstraction of $[\![\text{-}]\!]$ wrt $\mathcal{O}_t$ and wrt $\mathcal{O}_{t_\bot}$ means, respectively:

$$[\![e]\!] = [\![e']\!] \Leftrightarrow \forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$$

and

$$[\![e]\!] = [\![e']\!] \Leftrightarrow \forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e'])$$

where $\mathcal{P}'$ range over safe extensions, and $\mathcal{C}$ over $\mathcal{P}'$-contexts. Therefore, we must simply prove:

$$\forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e']) \Leftrightarrow \forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e'])$$

The $\Leftarrow$-implication is obvious. For the reverse implication $\Rightarrow$, assume

$$\forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$$

We will prove $\forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e]) \subseteq \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e'])$ (the other inclusion holds similarly). Assume $t \in \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e])$, which means $\mathcal{C}[e] \to^* e'$ for some e' with $t \sqsubseteq |e'|$. Let t' the result of substituting fresh variables for each occurrence of $\bot$ in $|e'|$, and t'' the result of substituting a new constant *bot* for each occurrence of $\bot$ in $|e'|$. We introduce a new function symbol f_t defined by the rule $f_t(t') \to t''$. We have $f_t(\mathcal{C}[e]) \to^* t''$, which means $t'' \in \mathcal{O}_t^{\mathcal{P}''}(f_t(\mathcal{C}[e]))$, where $\mathcal{P}''$ is the safe extension of $\mathcal{P}'$ made with *bot*, f_t. By hypothesis, $t'' \in \mathcal{O}_t^{\mathcal{P}''}(f_t(\mathcal{C}[e']))$, which means $f_t(\mathcal{C}[e']) \to^* e''$ for some e'' with $t'' \sqsubseteq |e''|$. As t'' is total by construction, it must be $t'' = |e''| = e''$, and therefore $f_t(\mathcal{C}[e']) \to^* t''$. Using the rewrite rule of f_t, it is not difficult to see that there must exist e''' with $\mathcal{C}[e'] \to^* e'''$ and $|e'| \sqsubseteq |e'''|$. But then $t \sqsubseteq |e'''|$, and therefore $t \in \mathcal{O}_{t_\bot}^{\mathcal{P}'}(\mathcal{C}[e'])$, as desired.

(b) The proof of equivalence of full abstraction wrt $\mathcal{O}_t$ and $\mathcal{O}_{t_\perp}$ proceeds as in *(a)*. It remains to prove that $[\![\_]\!]$ is fully abstract wrt $\mathcal{O}_t \Leftrightarrow$ is fully abstract wrt $\mathcal{O}_{true}$, for which we must see that

$$\forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e']) \Leftrightarrow \forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_{true}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_{true}^{\mathcal{P}'}(\mathcal{C}[e'])$$

where in this case $\mathcal{C}[e], \mathcal{C}[e']$ are restricted to be ground. The $\Rightarrow$-implication is obvious. For the reverse implication $\Leftarrow$, assume $\forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_{true}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_{true}^{\mathcal{P}'}(\mathcal{C}[e'])$. We will prove $\forall \mathcal{P}', \mathcal{C}.\ \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) \subseteq \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$ (the other inclusion holds similarly). Assume $t \in \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e])$, which means $\mathcal{C}[e] \to^* t$. Notice that, since $\mathcal{C}[e]$ is ground, t must also be ground. We introduce a new function symbol f_t defined by the rule $f_t(t) \to true$. We have $f_t(\mathcal{C}[e]) \to^* true$, which means $true \in \mathcal{O}_{true}^{\mathcal{P}''}(f_t(\mathcal{C}[e]))$, where $\mathcal{P}''$ is the safe extension of $\mathcal{P}'$ made with f_t. By hypothesis, $true \in \mathcal{O}_{true}^{\mathcal{P}''}(f_t(\mathcal{C}[e']))$, which means $f_t(\mathcal{C}[e']) \to^* true$. But, looking at the rewrite rule of f_t, this can only happen if $\mathcal{C}[e'] \to^* t$, that is, $t \in \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$. Notice how the groundness hypothesis is used: if t is not ground, the condition $f_t(\mathcal{C}[e']) \to^* true$ does not imply $\mathcal{C}[e'] \to^* t$, but only $\mathcal{C}[e'] \to^* t'$ for some instance t' of t.

Proof (Lemma 5). We assume a given $\mathcal{P}$. and mentions to P, P' are omitted in the notations below, since they can be deduced by the context. We reason by induction on the structure of st, noting first that, by definition of $[\![\_]\!]_S$ and Lemma 3, $st \in [\![e]\!]_S \Leftrightarrow st \in [\![\tilde{e}]\!] \Leftrightarrow e \succcurlyeq st$. Thus, the lemma can be reformulated as: $e \succcurlyeq st \Leftrightarrow f_{st}(e) \to^*_{\mathcal{P}'} \widehat{st}$

- $st \equiv \emptyset$: this case is obvious since $\emptyset \in [\![e]\!]_S, \forall e$ and $f_\emptyset(e) \to \langle\ \rangle_0 \equiv \widehat{\emptyset}, \forall e$.
- $st \equiv \{X\}$: then $e \succcurlyeq \{X\} \Leftrightarrow e \to^* X \Leftrightarrow f_{\{X\}}(e) \to^* X \equiv \widehat{\{X\}}$.
- $st \equiv \{c(st_1, \ldots, st_n)\}$: then $e \succcurlyeq \{c(st_1, \ldots, st_n)\} \Leftrightarrow \exists e_1, \ldots e_n.e \to^* c(e_1, \ldots, e_n)$ with $e_i \succcurlyeq st_i$ for each i. By IH, $e_i \succcurlyeq st_i \Leftrightarrow f_{st_i}(e_i) \to^* \widehat{st_i}$. On the other hand, looking at the program rule for $f_{\{c(st_1,\ldots,st_n)\}}$, we have also that $f_{\{c(st_1,\ldots,st_n)\}}(e) \to^* \widehat{c(st_1, \ldots, st_n)} (\equiv c(\widehat{st_1}, \ldots, \widehat{st_n})) \Leftrightarrow \exists e_1, \ldots e_n.e \to^* c(e_1, \ldots, e_n)$ with $f_{st_i}(e_i) \to^* \widehat{st_i}$.
- $st \equiv \{est_1, \ldots, st_n)\}$: then $e \succcurlyeq \{est_1, \ldots, st_n)\} \Leftrightarrow e \succcurlyeq \{est_i\} \forall i \Leftrightarrow_{IH} f_{\{est_i\}}(e) \to^* \widehat{\{est_i\}} \forall i$ (†). We must see that this condition is equivalent to $f_{\{est_1,\ldots,st_n)\}}(e) \to^* \{\{\widehat{est_1, \ldots, st_n)}\}\} \equiv \langle \widehat{\{est_1\}}, \ldots, \widehat{\{est_n\}} \rangle_n$ (‡). That † $\Rightarrow$ ‡ is clear, just starting by the step $f_{\{est_1,\ldots,st_n)\}}(e) \to \langle f_{\{est_1\}}(e), \ldots, f_{\{est_n\}}(e) \rangle_n$ and using † in each component of the tuple. For ‡ $\Rightarrow$ †, notice that any rewrite sequence $f_{\{est_1,\ldots,st_n)\}}(e) \to^* \{\{\widehat{est_1, \ldots, est_n)}\}\} \equiv \langle \widehat{\{est_1\}}, \ldots, \widehat{\{est_n\}} \rangle_n$ must start with the form

$$f_{\{est_1,\ldots,st_n)\}}(e) \to^* f_{\{est_1,\ldots,st_n)\}}(e') \to \langle f_{\{est_1\}}(e'), \ldots, f_{\{est_n\}}(e') \rangle_n \to^* \langle \widehat{\{est_1\}}, \ldots, \widehat{\{est_n\}} \rangle_n$$

where $e \to *e'$ and $f_{\{est_i\}}(e') \to^* \widehat{\{est_i\}}$ for each i. But this leads to $f_{\{est_i\}}(e) \to^* \widehat{\{est_i\}}$ for each i, which is precisely (†).

Bibliography

[ABB+98] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised 5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[ABC06] S. Antoy, D. Brown, and S. Chiang. On the Correctness of Bubbling. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications (RTA'06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2006.

[ABC07] S. Antoy, D. Brown, and S. Chiang. Lazy Context Cloning for Non-deterministic Graph Rewriting. *Electronic Notes in Theoretical Computer Science*, 176(1):61–70, 2007. Proceedings of the Third International Workshop on Term Graph Rewriting (Termgraph'06).

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[ACE+03] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In *12th International Workshop on Logic Based Program Synthesis and Tranformation (LOPSTR'02), Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2003.

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proceedings of the 21nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'94)*, pages 268–279. ACM, 1994.

[AF97] Z. M. Ariola and M. Felleisen. The Call-By-Need Lambda Calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[AFM+95] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A Call-by-need Lambda Calculus. In *Proceedings of the 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'95)*, pages 233–246, 1995.

[AFRV93] M. Alpuente, M. Falaschi, M.J. Ramis, and G. Vidal. Narrowing Approximations as an Optimization for Equational Logic Programs. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, volume 714 of *Lecture Notes in Computer Science*, pages 391–409. Springer, 1993.

[AH00] S. Antoy and M. Hanus. Compiling Multi-paradigm Declarative Programs Into Prolog. In *Proceedings of the Third International Workshop on Frontiers of Combining Systems (FroCos'00)*, volume 1794 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2000.

[AH02] S. Antoy and M. Hanus. Functional Logic Design Patterns. In Zhenjiang Hu and Mario Rodríguez-Artalejo, editors, *Proceedings of the 6th Fuji International Symposium on Functional and Logic Programming (FLOPS'02)*, volume 2441 of *Lecture Notes in Computer Science*, pages 67–87. Springer, 2002.

[AH06] S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *22nd International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2006.

[AHH+05] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[AIM02] S. Antoy, Pascual Julián Iranzo, and Bart Massey. Improving the Efficiency of Non-deterministic Computations. *Electronic Notes in Theoretical Computer Science*, 64, 2002.

[Ant92] S. Antoy. Definitional Trees. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming (ALP'92)*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 1992.

[Ant97] S. Antoy. Optimal Non-deterministic Functional Logic Computations. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming (ALP'97)*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1997.

[Ant05] S. Antoy. Evaluation Strategies for Functional Logic Programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[AR01] Puri Arenas-Sánchez and Mario Rodríguez-Artalejo. A General Framework for Lazy Functional Logic Programming With Algebraic Polymorphic Types. *Theory and Practice of Logic Programming*, 2(1):185–245, 2001.

[Arm07] Joe Armstrong. A History of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[AT99] S. Antoy and Andrew P. Tolmach. Typed Higher-order Narrowing without Higher-order Strategies. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 335–353. Springer, 1999.

[AZTML08] Abdallah D. Al Zain, Phil W. Trinder, Greg J. Michaelson, and Hans-Wolfgang Loidl. Evaluating a High-level Parallel Language (GpH) for Computational GRIDs. *IEEE Trans. Parallel Distrib. Syst.*, 19(2):219–233, 2008.

[BCL86] G. Berry, PL. Curien, and JJ. Levy. Full Abstraction for Sequential Languages: the State of the art. In *Algebraic methods in semantics*, pages 89–132. Cambridge University Press, New York, NY, USA, 1986.

[BEG+87] H P Barendregt, M C J D Eekelen, J R W Glauert, J R Kennaway, M J Plasmeijer, and M R Sleep. Term Graph Rewriting. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 141–158, London, UK, 1987. Springer-Verlag.

[BEØ93] D. Bert, R. Echahed, and M. Østvold. Abstract rewriting. In *Proceedings of the Third International Workshop on Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 1993.

[BH07] Bernd Braßel and Frank Huch. On a Tighter Integration of Functional and Logic Programming. In *Proceedings of the 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, volume 4807 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2007.

[BHH04] B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[BJM00] A. Bouhoula, J. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[Ble93] Guy E. Blelloch. NESL: A Nested Data-parallel Language (Version 2.6). Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, United Kingdom, 1998.

[Bou81] Gérard Boudol. Une Semantique Pour les Arbres non Deterministes. In *Proceedings of the 6th Colloquium on Trees in Algebra and Programming (CAAP '81)*, volume 112 of *Lecture Notes in Computer Science*, pages 147–161, London, UK, 1981. Springer.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[CLLF04] José Miguel Cleva, Javier Leach, and Francisco Javier López-Fraguas. A Logic Programming Approach to the Verification of Functional-logic Programs. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 9–19. ACM, 2004.

[CMP07] M. Clavel, J. Meseguer, and M. Palomino. Reflection in Membership Equational Logic, Many-sorted Equational Logic, Horn Logic With Equality, and Rewriting Logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach*, volume 1: Foundations of *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 163–245. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[Cou90] Brouno Courcelle. *Graph Rewriting: an Algebraic and Logic Approach*, volume B: Formal Models and Semantics of *Handbook of Theoretical Computer Science*, pages 193–242. Elsevier, 1990.

[CP06] J.M. Cleva and I. Pita. Verification of CRWL programs with rewriting logic. *Journal of Universal Computer Science*, 12(11):1594–1617, 2006.

[CPR06] M. Clavel, M. Palomino, and Adrián Riesco. Introducing the ITP Tool: a Tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006.

[CRR09] Rafael Caballero, Mario Rodríguez-Artalejo, and Carlos A. Romero-Díaz. Qualified Computations in Functional Logic Programming. In *Proceedings of the 25th International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 449–463. Springer, 2009.

[CSe06] Rafael Caballero and Jaime Sánchez-Hernández (eds.). TOY: A Multiparadigm Declarative Language, Version 2.2.3. Technical report, Universidad Complutense de Madrid, July 2006.

[dDL07] Javier de Dios-Castro and Francisco Javier López-Fraguas. Extra Variables can be Eliminated From Functional Logic Programs. *Electronic Notes in Theoretical Computer Science 188*, pages 3–19, 2007.

[DEL05] F. Durán, S. Escobar, and S. Lucas. On-demand Evaluation for Maude. In *Proceedings of the 5th International Workshop on Rule-Based Programming (RULE'04)*, Electronic Notes in Theoretical Computer Science, pages 263–284, 2005.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall International (UK) Ltd., Upper Saddle River, NJ, USA, 1997.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'82)*, pages 207–212, New York, NY, USA, 1982. ACM.

[dMvEP01] Maarten de Mol, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Theorem Proving for Functional Programmers. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'02)*, volume 2312 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2001.

[DS93] Iréne Durand and Bruno Salinier. Constructor Equivalent Term Rewriting Systems. *Information Processing Letters*, 47(3):131–137, 1993.

[EFS+09] Sonia Estévez-Martín, Antonio J. Fernández, Fernando Sáenz-Pérez, Teresa Hortalá-González, Mario Rodríguez-Artalejo, and Rafael del Vado-Vírseda. On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in *CFLP*. *Theory and Practice of Logic Programming*, 9:4:415–527, July 2009.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic Approaches to Graph Transformation, Part II: Single Pushout Approach and Comparison With Double Pushout Approach*, volume 1: Foundations of *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 247–312. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[EJ97] R. Echahed and J.-C. Janodet. On Constructor-based Graph Rewriting Systems. Research Report 985-I, IMAG, 1997. Available at ftp://ftp.imag.fr/pub/LEIBNIZ/ATINF/c-graph-rewriting.ps.gz.

[EJ98] R. Echahed and J.-C. Janodet. Admissible Graph Rewriting and Narrowing. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 325 – 340, Manchester, June 1998. MIT Press.

[EMT05] S. Escobar, J. Meseguer, and P. Thati. Natural Narrowing for General Term Rewriting Systems. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA'05)*, pages 279–293, 2005.

[Esc03] S. Escobar. Refining Weakly Outermost-needed Rewriting and Narrowing. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declaritive Programming (PPDP'03)*, pages 113–123. ACM, 2003.

[Esc04] S. Escobar. Implementing Natural Rewriting and Narrowing Efficiently. In *Proceedings of the 7th Fuji International Symposium on Functional and Logic Programming (FLOPS'04)*, volume 2998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2004.

[FBK05] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005.

[FKS09] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely Functional Lazy Non-deterministic Programming. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (ICFP '09)*, pages 11–22, New York, NY, USA, 2009. ACM.

[FN97] Kokichi Futatsugi and Ataru T. Nakagawa. An Overview of CAFE Specification Environment - an Algebraic Approach for Creating, Verifying, and Maintaining Formal Specifications Over Networks. In *Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM'97)*, page 170, 1997.

[FRR+07] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: a Heterogeneous Parallel Language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming (DAMP '07)*, pages 37–44, New York, NY, USA, 2007. ACM.

[GHLR96] Juan Carlos González-Moreno, Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. A Rewriting Logic for Declarative Programming. In *Proc. European Symposium on Programming (ESOP'96)*, volume 1058 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 1996.

[GHLR99] Juan Carlos González-Moreno, Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[GHR92] Juan Carlos González-Moreno, Teresa Hortalá-González, and Mario Rodríguez-Artalejo. On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic on Computer Science Logic (CSL'92)*, volume 702 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 1992.

[GHR97] Juan Carlos González-Moreno, Teresa Hortalá-González, and Mario Rodríguez-Artalejo. A Higher Order Rewriting Logic for Functional Logic Programming. In *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

[GHR01] Juan Carlos González-Moreno, Teresa Hortalá-González, and Rodríguez-Artalejo, Mario. Polymorphic Types in Functional Logic Programming. In *Journal of Functional and Logic Programming*, volume 2001/S01, pages 1–71, 2001.

[Gon93] Juan Carlos González-Moreno. A Correctness Proof for Warren's HO into FO Translation. In *Proceedings of the 8th Italian Conference on Logic Programming (GULP'93)*, pages 569–584, 1993.

[Han94] M. Hanus. The Integration of Functions Into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han06] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Version 0.8.2). Available at http://www.informatik.uni-kiel.de/~curry/report.html, March 2006.

[Han07] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007.

[Han09] M. Hanus (ed.). *PAKCS 1.9.2, The Portland Aachen Kiel Curry System, User manual*, 2009. http://www.informatik.uni-kiel.de/~pakcs/Manual.pdf.

[Has10a] Haskell.org. *The List monad*, 2010. http://www.haskell.org/all_about_monads/html/listmonad.html.

[Has10b] Haskell.org. *The State monad*, 2010. http://www.haskell.org/all_about_monads/html/statemonad.html.

[Hen80] Peter Henderson. Purely Functional Operating Systems. In *Functional programming and its applications*, pages 177–192. Cambridge University Press, 1980.

[HH04] Mercedes Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo*. PhD thesis, Universidad Complutense de Madrid, 2004.

[Hin00] Ralf Hinze. Deriving Backtracking Monad Transformers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP'00)*, pages 186–197, 2000.

[HL01] M. Hanus and S. Lucas. An Evaluation Semantics for Narrowing-Based Functional Logic Languages. *Journal of Functional and Logic Programming*, 2001(2), 2001.

[HM95] John Hughes and Andrew Moran. Making Choices Lazily. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 108–119, 1995.

[HMJH08] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. *Communications of the ACM*, 51(8):91–100, 2008.

[HO90] John Hughes and John O'Donnell. Expressing and Reasoning About Nondeterministic Functional Programs. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 308–328, London, UK, 1990. Springer.

[HP99] M. Hanus and Christian Prehofer. Higher-order Narrowing With Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[Hul80] J.M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th Conference on Automated Deduction (CADE'80)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

[Hus93] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[KSFS05] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (ICFP'05)*, pages 192–203. ACM, 2005.

[KSS98] Arne Kutzner and Manfred Schmidt-Schauß. A Non-deterministic Call-by-need Lambda Calculus. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 324–335, 1998.

[Lau93] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM, 1993.

[LF07] Francisco Javier López Fraguas. Curry mailing list: Re: New pakcs release (version 1.8.0). http://www.informatik.uni-kiel.de/~curry/listarchive/0497.html, March 2007.

[LLR93] Rita Loogen, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, volume 714 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 1993.

[LM99] Søren B. Lassen and Andrew Moran. Unique Fixed Point Induction for McCarthy's Amb. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, volume 1672 of *Lecture Notes in Computer Science*, pages 198–208. Springer, 1999.

[LMJT07] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight Concurrency Primitives for GHC. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop (Haskell'07)*, pages 107–118, New York, NY, USA, 2007. ACM.

[LMR09a] Francisco Javier López-Fraguas, Stephan Merz, and Juan Rodríguez-Hortalá. A Formalization of the Semantics of Functional-Logic Programming in Isabelle. *Computing Research Repository CoRR*, abs/0908.0494, 2009. Emerging trends section (category B) of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09). Available at http://arxiv.org/abs/0908.0494.

[LMR09b] Francisco Javier López-Fraguas, Stephan Merz, and Juan Rodríguez-Hortalá. A Formalization of the Semantics of Functional-Logic Programming in Isabelle (Extended version). Technical Report SIC-4-09, Universidad Complutense de Madrid, 2009.

[LMR09c] Francisco Javier López-Fraguas, Stephan Merz, and Juan Rodríguez-Hortalá. *IsabelleCrwl*, 2009. https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/IsabelleCrwl.

[LMR10] Francisco Javier López-Fraguas, Enrique Martin-Martin, and Juan Rodríguez-Hortalá. New results on type systems for functional logic programming. In *18th International Workshop on Functional and (Constraint) Logic Programming WFLP'09, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2010. To appear.

[LOP05] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[LR10] Francisco Javier López-Fraguas and Juan Rodríguez-Hortalá. The Full Abstraction Problem for Higher Order Functional-Logic Programs. *Computing Research Repository CoRR*, abs/1002.1833, 2010. Proceedings of the 19th Workshop on Logic-based methods in Programming Environments (WLPE'09). Available at http://arxiv.org/abs/1002.1833.

[LRS07a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Equivalence of Two Formal Semantics for Functional Logic Programs. *Electronic Notes in Theoretical Computer Science*, 188:117–142, 2007. Proceedings of the Sixth Spanish Conference on Programming and Languages (PROLE'06).

[LRS07b] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Simple Rewrite Notion for Call-time Choice Semantics. In *Proceedings of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM, 2007.

[LRS07c] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Bundles Pack Tighter Than Lists. In *Draft Proceedings of Eighth Symposium on Trends in Functional Programming (TFP'07)*, pages XXIV–1–XXIV–16, 2007.

[LRS08a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Rewriting and Call-Time Choice: The HO Case. In *Proceedings of the 9th Fuji International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2008.

[LRS08b] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. *RtToyCt*. Universidad Complutense de Madrid, 2008. https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RtToy#Call-timechoicebasedvariant.

[LRS09a] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Flexible Framework for Programming with Non-deterministic Functions. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, pages 91–100. ACM, 2009.

[LRS09b] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Fully Abstract Semantics for Constructor Systems. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2009.

[LRS09c] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A Lightweight Combination of Semantics for Non-deterministic

Functions. *Computing Research Repository (CoRR)*, abs/0903.2205, 2009. Proceedings of the 18th Workshop on Logic-based methods in Programming Environments (WLPE'08). Available at http://arxiv.org/abs/0903.2205.

[LRS09d] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. Narrowing for First Order Functional Logic Programs with Call-Time Choice Semantics. In *Proceedings of the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP'07) and 21st Workshop on (Constraint) Logic Programming (WLP'07), Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 206–222. Springer, 2009.

[LRS09e] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. *RtToyRt*. Universidad Complutense de Madrid, 2009. https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/RtToy#Run-timechoicebasedvariant.

[LRV04] Francisco Javier López-Fraguas, Mario Rodríguez-Artalejo, and Rafael del Vado-Vírseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *Proceeding of the 6th ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 43–54. ACM, 2004.

[LRV07] Francisco Javier López-Fraguas, Mario Rodríguez-Artalejo, and Rafael del Vado-Vírseda. A new Generic Scheme for Functional Logic Programming With Constraints. *Higher-Order and Symbolic Computation*, 20(1-2):73–122, 2007.

[LS99] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247. Springer, 1999.

[Lux07] W. Lux. *MCC 0.9.11, The Münster Curry Compiler, User's Guide*, 2007. http://danae.uni-muenster.de/~lux/curry/user.html.

[Lux09] W. Lux. Curry mailing list: Type-classes and call-time choice vs. run-time choice. http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html, August 2009.

[McC63] John McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[Mil91] Dale Miller. A Logic Programming Language With Lambda-abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mit96] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, Cambridge, MA, USA, 1996.

[MM02] N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[MOW98] J. Maraist, M. Odersky, and P. Wadler. The Call-by-need Lambda Calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[NAR07] Matthew Naylor, Emil Axelsson, and Colin Runciman. A Functional-logic Library for Wired. In *Proceedings of the 2007 ACM SIGPLAN workshop on Haskell (Haskell'07)*, pages 37–48. ACM, 2007.

[NMI95] Koichi Nakahara, Aart Middeldorp, and Tetsuo Ida. A Complete Narrowing Calculus for Higher-order Functional Logic Programming. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'95)*, pages 97–114, 1995.

[Nys96] Sven-Olof Nyström. There is no Fully Abstract Fixpoint Semantics for Nondeterministic Languages With Infinite Computations. *Information Processing Letters*, 60(6):289–293, 1996.

[O'D85] Michael J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[Ohl02] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 2002.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International (UK) Ltd., 1987.

[PJ03] S. (ed.) Peyton Jones. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge Univ. Press, 2003.

[PJ08] Simon Peyton Jones. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS '08)*, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[Pla95] Marinus J. Plasmeijer. CLEAN: a Programming Environment Based on Term Graph Rewriting. *Electronic Notes in Theoretical Computer Science*, 2, 1995.

[Plo75] Gordon D. Plotkin. Call-by-name, Call-by-Value and the Lambda-Calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[Plo77] Gordon D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5(3):225–255, 1977.

[Plu99] D. Plump. *Term Graph Rewriting*, volume 2: Applications, Languages, and Tools of *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 3–61. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[Pol09] David Pollak. *Beginning Scala*. Apress, Berkely, CA, USA, 2009.

[PvE93] R. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[Rep91] John H. Reppy. CML: A Higher-order Concurrent Language. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI '91)*, pages 293–305, New York, NY, USA, 1991. ACM.

[Rod01] Mario Rodríguez-Artalejo. Functional and Constraint Logic Programming. In *Revised Lectures of the 1999 International Summer School on Constraints in Computational Logics (CCL'99)*, volume 2002 of *Lecture Notes in Computer Science*, pages 202–270. Springer, 2001.

[Rod08] Juan Rodríguez-Hortalá. A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems. In *Proceedings of the 28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 328–339, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Rod09] Juan Rodríguez-Hortalá. Curry mailing list: Re: Type-classes and call-time choice vs. run-time choice. http://www.informatik.uni-kiel.de/~curry/listarchive/0801.html, August 2009.

[RR09a] Adrián Riesco and Juan Rodríguez-Hortalá. *A natural implementation of Plural Semantics in Maude*. Universidad Complutense de Madrid, 2009. https://gpd.sip.ucm.es/trac/gpd/wiki/PluralSemantics/Maude.

[RR09b] Adrián Riesco and Juan Rodríguez-Hortalá. A natural implementation of Plural Semantics in Maude (demonstration). *Electronic Notes in Theoretical Computer Science*, 2009. To appear in Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications (LDTA'09).

[RR10] Adrián Riesco and Juan Rodríguez-Hortalá. Programming with Singular and Plural Non-deterministic Functions. In *Proceedings of the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM'10)*, pages 83–92. ACM, 2010.

[Sal95] Bruno Salinier. *Simulation de systèmes de réécriture de termes par des systèmes constructeurs*. PhD thesis, Université de Bordeaux 1, Talence, France, 1995.

[SH04] Jaime Sánchez-Hernández. *Una aproximación al fallo constructivo en programación declarativa multiparadigma*. PhD thesis, Universidad Complutense de Madrid, 2004.

[Sör83] Holström Sören. PFL: A Functional Language for Parallel Programming and Its Implementation. In *Declarative Programming Worshop at University College*, London, UK, April 11–13, 1983. Also: Technical Report 83.03 R, Department of Computer Science,Chalmers University of Technology, Göteborg, Sweden.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[SS92] H. Søndergaard and P. Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, 1992.

[SSH00] Manfred Schmidt-Schauß and Michael Huber. A Lambda-Calculus with letrec, case, constructors and non-determinism. *Computing Research Repository (CoRR)*, cs.PL/0011008, 2000.

[Stä98] Robert F. Stärk. The Theoretical Foundations of Lptp (a Logic Program Theorem Prover). *Journal of Logic Programming*, 36(3):241–269, 1998.

[Sto84] W. Stoye. A New Scheme for Writing Functional Operating Systems. Technical Report 56, University of Cambridge Computer Laboratory, 1984.

[Str06] Thomas Streicher. *Domain-theoretic Foundations of Functional Programming*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2006.

[TeR03] TeReSe. *Term Rewriting Systems*, volume No. 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[Tha85] Satish R. Thatte. On the Correspondence Between two Classes of Reduction Systems. *Information Processing Letters*, 20(2):83–85, 1985.

[TS09] René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.

[Vad03] Rafael del Vado-Vírseda. A Demand-driven Narrowing Calculus With Overlapping Definitional Trees. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declaritive Programming (PPDP'03)*, pages 213–227. ACM, 2003.

[Vad07] Rafael del Vado-Vírseda. A Higher-order Demand-driven Narrowing Calculus With Definitional Trees. In *Proceedings of the 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, volume 4711 of *Lecture Notes in Computer Science*, pages 169–184. Springer, 2007.

[Vad09] Rafael del Vado-Vírseda. A Higher-order Logical Framework for the Algorithmic Debugging and Verification of Declarative Programs. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 49–60. ACM, 2009.

[vdBMR02] Mark van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen. The ELAN Environment: a Rewriting Logic Environment Based on ASF+SDF Technology—System Demonstration. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002. Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA'02).

[Wad85] P. Wadler. How to Replace Failure by a List of Successes. In *Proceedings of Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer, 1985.

[War82] David H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., 1982.

[WB89] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, New York, NY, USA, 1989. ACM.

[Wik87] Ake Wikström. *Functional Programming Using Standard ML*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1987.

[Wik10] Wikibooks. *Haskell/MonadPlus*, 2010. http://en.wikibooks.org/wiki/Haskell/MonadPlus.

[WP06] Markus Wenzel and Lawrence C. Paulson. Isabelle/Isar. In *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 41–49. Springer, 2006.